REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-01-

0197

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPO...
	2/27/01	Final Technical 9/1/96-8/31/00

4. TITLE AND SUBTITLE
CFD Simulation and Visual Analysis of Complex Time-Dependent Flight Vehicle Flow Fields

5. FUNDING NUMBERS
F49620-96-1-0441

6. AUTHOR(S)
David L. Marcum

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Mississippi State University
Box 6156
Mississippi State, MS 39762

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research
801 North Randolph Street, Room 732
Arlington, VA 22203-1977

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author and should not be construed as an official Air Force position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The primary objective of this research project was to produce a research capability to perform detailed CFD simulations and Scientific Visualization analysis of unsteady, three-dimensional, compressible, viscous flow fields about flight vehicle configurations of interest to the Department of Defense (DoD). The target applications include, but are not limited to, separation of single or multiple stores from aircraft or missiles, launch vechicle or missile booster separation, separation of crew escape modules or ejection seats, separation of fairings, etc. from missile systems, and maneuvering aircraft or missiles. Target flow conditions include, compressible flow regimes with vehicles operating at Mach numbers from high subsonic to moderate hyper-sonic, laminar and turbulent viscous flow, and flow of gases in chemical and thermal equilibrium.

14. SUBJECT TERMS
computational fluid dynamics, unstructured grid generation, scientific visualization

15. NUMBER OF PAGES
183

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CFD Simulation & Visual Analysis of Complex Time-Dependent Flight Vehicle Flow Fields

Submitted to the Air Force Office of Scientific Research
in Fulfillment of the Requirements for
DEPSCOR95 Grant F49620-96-1-0441

Mississippi State University
Mississippi State, Mississippi

TABLE OF CONTENTS

CHAPTER Page

APPENDIX Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Recent advances in grid generation, solution algorithms, scientific visualization, and computer architecture have made it possible to simulate and analyze flow fields about increasingly complex flight vehicles using Computational Fluid Dynamics (CFD). However, there is still a significant difference between reality and the overall complexity of the simulations. Computational models of flight vehicles that are often labeled as complex or complete are usually simplified approximations to the real vehicle. Simulation of unsteady, viscous flow fields about vehicles with real operating conditions, such as maneuvering vehicles, varying engine conditions, moving or separating components, and separating stores, are typically considered too demanding for current technology. In addition, tools available for Scientific Visualization analysis of the data generated by time-dependent simulations are not adequate. The underlying fluid mechanics of complex time-dependent flow fields is not well understood, and usable computational simulation and analysis tools could provide significant insight. There is a real need for a capability to simulate and analyze complex flow fields about flight vehicles with realistic geometry and operating conditions. This research addresses this need for the case of flight vehicle configurations with separating stores, moving or separating components, and components during maneuvers. These applications are of importance to many DoD agencies and the aerospace industry.

The primary objective of this research project was to produce a research capability to perform detailed CFD simulations and Scientific Visualization analysis of unsteady, three-dimensional, compressible, viscous flow fields about flight vehicle configurations of interest to the Department of Defense (DoD). The target applications include, but are not limited to, separation of single or multiple stores from aircraft or missiles, launch vehicle or missile booster separation, separation

of crew escape modules or ejection seats, separation of fairings, etc. from missile systems, and maneuvering aircraft or missiles. Target flow conditions include, compressible flow regimes with vehicles operating at Mach numbers from high sub-sonic to moderate hyper-sonic, laminar and turbulent viscous flow, and flow of gases in chemical and thermal equilibrium.

At Mississippi State University, the Computational Fluid Dynamics Laboratory (CFD Lab) at the National Science Foundation Engineering Research Center for Computational Field Simulation conducts an extensive program of application-driven basic and applied research on computer-based simulation and design methodology that encompasses grid generation, flow solvers and visualization using both structured and unstructured grid topologies, scalable parallel computing, technology demonstrations for leading-edge problems, and integrated simulation systems for design environments. The CFD Lab has brought together a group of individuals from a variety of engineering and computational disciplines for the common goal of creating a software environment that is capable of completing all tasks required for CFD analysis[1]. The creation of this multidisciplinary environment has brought about the development of a rich suite of tools that take the problem of solving complex physics on complex configurations from geometry to grid to solution through to visualization [2], [3], [4]. This blend of individuals from a variety of disciplines is a natural environment for conducting research on the complex time-dependent problems for this grant. The capabilities developed under this grant are a direct result of having worked as a cross-disciplinary team created to solve and analyze numerical simulations of complex flow on complex geometries.

As a capstone example of the capability created within the scope of this grant, a simulation of a strap-on booster separating from a Delta II launch vehicle was performed. In the overall simulation, flow about the complete configuration was initially modeled. Next, the strap-on booster by itself was modeled just after separation. During this portion of the time-accurate simulation, the strap-on booster tumbles. An integrated six-degrees-of-freedom (6DOF) model determines the strap-on booster kinematics based on the aerodynamic loadings. This simulation required significant advances in the areas of unstructured grid generation, unsteady solution

algorithms, and parallel implementations. Scientific visualization was then used to analyze the flow field data from the simulation. The visual analysis of the flow field was done using a suite of scientific visualization tools developed in part under this grant. Finally, a movie was generated which depicts the flow field physics and strap-on booster kinematics. The movie tracks both the motion and flow field variables on the strap-on booster surface and in the surrounding field.

The research capabilities produced from this grant have provided insight into the fluid mechanics of complex time-dependent flow fields for flight vehicles with real operating conditions and has and will guide future research in this area. These tools significantly advance the state-of-the-art in CFD, Scientific Visualization, and the overall simulation and analysis capability available to DoD agencies and the aerospace industry. As an example of the capabilities guiding future research, technology developed in part under this grant has also been applied to capstone simulations of submarine maneuvers and gust response of a tilt-rotor aircraft.

The following chapters discuss the various fields of study that were investigated to meet the objectives of this grant. Chapter 2 provides a background on the grid generation algorithms that were used to generate the static portions of the grids. Chapter 3 provides the details on the relative body motion calculations and is a companion chapter with chapter 4, discussing the 6DOF model. Chapter 5 provides the details for the computational methodology. Given in chapters 6 and 7 is an overview of the visualization tools developed to animate the time-dependent unsteady simulation. Finally, chapter 8 discusses the movie and pertinent results produced from this research.

CHAPTER II

GRID GENERATION

2.1 Introduction

Unstructured grid technology is a promising approach offering geometric flexibility for handling of both complex geometry and physics. As such, it can provide a powerful capability for accurately and efficiently computing complex flow fields about realistic aerospace configurations. Several unstructured grid generation and flow solver procedures have been developed and successfully demonstrated for inviscid flow about complex configurations. For isotropic elements, existing procedures are robust and capable of generating high-quality grids efficiently. For anisotropic elements in viscous flow applications, further improvements in efficiency, robustness, and quality of unstructured grid generation procedures are needed. In this chapter, a grid generation procedure is presented which offers the potential for overall improved performance and quality.

The most common approach used to generate anisotropic unstructured grids is to use a layered approach and generate points along normals from solid boundaries. Unstructured grid generation for viscous applications have been developed using a modified advancing-front method by Hassan, et al [5], a semi-structured approach by Lohner [6], advancing-normal by Marcum [7], and advancing-layers by Pirzadeh [8]. Hybrid methods for prismatic/tetrahedral grid generation have been developed by Kallinderis [9] and Sharov and Nakahasi [10]. While all of these methods differ in how points and elements are generated, they all produce very structured and aligned elements adjacent to solid boundaries and use isotropic tetrahedral elements outside of the anisotropic or boundary-layer region. Use of prismatic elements within the anisotropic region can reduce subsequent memory and CPU requirements for the flow solver without any loss of accuracy.

The goal of the present work is to modify the existing advancing-normal and advancing-front local-reconnection method [7] for efficient generation of high-quality, mixed element type

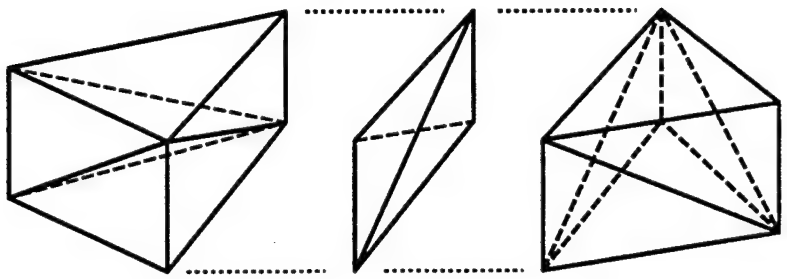


Figure 2.1: Trapped sliver element between prismatic groups of tetrahedral elements.

unstructured grids for viscous flow applications. While this method has many advantages, the local-reconnection process is unable to remove trapped sliver elements. Such elements can be formed between prismatic groups of elements as shown in Figure 2.1. Local-reconnection or connectivity optimization can not remove these trapped slivers as they represent a local minimum state that cannot be removed without reconnecting a potentially very large number of elements. An alternative is to use the same point placement strategy and discard the connectivity in favor of a hybrid approach. With a hybrid approach the element connectivity is directly implied. Also, the elements can be recovered as either all tetrahedra or a mixture of five and six node pentahedra and tetrahedra. This combined approach retains most of the generality of the original and is very efficient.

2.2 Unstructured Grid Generation Procedure

The approach used in the present work uses the advancing-normal point placement algorithm [7] to generate points within the anisotropic region. The advancing-front/ local-reconnection (AFLR) procedure [11],[12] is used to generate tetrahedral elements in the isotropic region. The basic steps in the overall procedure are listed below.

1. Generate a boundary surface grid.
2. Generate a volume triangulation of the boundary points. No boundary recovery is required.
3. Create new points using advancing-normal point placement.
4. Attach new points to the existing triangulation for searching and checking.
5. Create a new boundary surface grid using the inflated surface from advancing-normal point placement.
6. Use AFLR to generate an isotropic tetrahedral element grid for the remaining regions.
7. Merge anisotropic and isotropic regions. Element connectivity within the anisotropic region is directly determined from the point ordering.

2.2.1 Advancing-Normal Anisotropic Grid Generation

With advancing-normal type point placement for high-aspect-ratio elements, the standard AFLR procedure [7] does produce sliver elements of the type shown in Figure 2.1. These elements are generated only in regions of high-aspect-ratio elements with a very structured alignment. Elimination of these elements with local-reconnection is not feasible. There may be no nearby optimization path which produces a better connectivity. The problem is inherently due to the very structured nature of the grid in these regions. Only a limited set of possible triangulations, that do not contain sliver elements, exists for a set of tetrahedra aligned in prismatic groups. A modified process is proposed here which eliminates the sliver problem and retains the generality and efficiency of the original procedure. In the present approach, local-reconnection is not used to determine the connectivity in these regions, Instead, the connectivity is directly determined by the order in which points were generated. This produces a very

structured connectivity and allows the elements created to either be all tetrahedra or of mixed type.

The basic steps in the modified procedure are listed below.

1. Determine a normal vector at each active boundary-layer point. Initially the normals are based solely on the original boundary surface geometry. As the generation advances the normals are generated using the geometry of the outer layer of the boundary-layer grid.

2. Smooth the normal vectors with a weighting dependent upon the distance from the boundary. Initially the normals are unsmoothed. At the estimated end of the boundary-layer region full smoothing is applied. The end of the boundary-layer region is based on a estimate of where the element aspect ratio will be near isotropic.

3. Generate new points one layer at a time. New points are created along the normal vector with the normal spacing determined using geometric growth from the boundary surface. Generation of new points along a normal is shown in Figure 2.2

4. Check distance between new points and surrounding element quality. The volume triangulation is used to efficiently check nearby points. As boundary-layers merge new points may be too close and advancement should terminate locally. A new point is rejected if the distance between it and any nearby new (or existing) point is less than a preset fraction of the local element length scale. Boundary-layer advancement is terminated locally if a new point is rejected.

5. New points are also rejected if any of the surrounding elements that they may produce fail a quality check (maximum angle < 160 deg.). Boundary-layer advancement is terminated locally if a new point is rejected for quality.

6. Active points can become isolated as boundary-layer advancement is locally terminated. This can be prevented by rejecting a new point and terminating local advancement if more than some fraction of its neighbors have been terminated.

7. Check element aspect-ratio. As the grid advances and the normal spacing increases the element aspect-ratio will eventually be isotropic. Boundary-layer advancement is terminated locally when the aspect-ratio on the next layer would be greater than unity.

8. Attach accepted new points to the volume triangulation. New points are connected and attached to the existing element that contains them.

9. Generate a new boundary surface grid by inflating the previous surface at points that have continued to advance.

10. Repeat steps 1 through 9 until no new points are accepted.

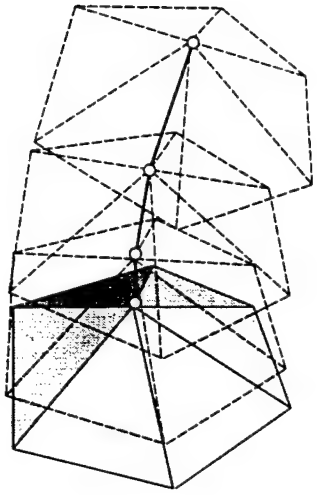


Figure 2.2: Advancing-normal point placement along normal line from boundary surface.

2.2.2 Normal Spacing

The normal spacing is determined using accelerated geometric growth. The initial normal spacing can be specified globally or at each boundary point. Standard geometric growth is used with an accelerated growth factor. The normal spacing is determined from

$$\Delta s_{n+1} = \alpha_n \Delta s_n \tag{2.1}$$

$$\alpha_{n+1} = min(\beta \alpha_n, \alpha_{max}) \tag{2.2}$$

where Δs_n is the normal spacing for layer n, α_n is the growth factor for layer n, α_{max} is the maximum allowable growth factor, and β is the growth acceleration factor.

2.2.3 Boundary Normal Vectors

A boundary normal vector is required for the advancing-normal procedure. This normal is determined on the inflated surface during each pass. On the first pass the surface is the same as the original boundary surface. As only the boundary face normals are unique (since planar faces are used), some form or averaging or optimization procedure must be used to obtain tne normal vector at nodes. Weighted averages can produce variations in normals due purely to topology or local face area differences. In the present work, a least-squares optimization procedure is used to eliminate those variations. An error function is defined as

$$e_j = I - b_i \bullet n_j \tag{2.3}$$

where e_j is the error function for face j, n_j is the face unit normal vector for face j, b_i is the node unit normal vector for node i. Node and face normals are shown in Figure 2.3. The error function is also directly related to the volume of the element that will be produced from a given face. Minimizing the error function also maximizes the element volume. Least-squares optimization can be used to find b_i such that $\sum e_j^2$ is minimized. The resulting equations are

$$\sum[(b_i \bullet n_j)n_j^x] = \sum n_j^x \tag{2.4}$$

$$\sum[(b_i \bullet n_j)n_j^y] = \sum n_j^y \tag{2.5}$$

$$\sum[(b_i \bullet n_j)n_j^z] = \sum n_j^z \tag{2.6}$$

where $\sum$ denotes the sum over all faces surrounding node i and n_j^x, n_j^y, and n_j^z are the x, y, and z components of the unit normal vector for face j.

2.2.4 Element Connectivity

The element connectivity for the points created using advancing-normal point placement is determined directly from the order in which they were created. The initial point ordering

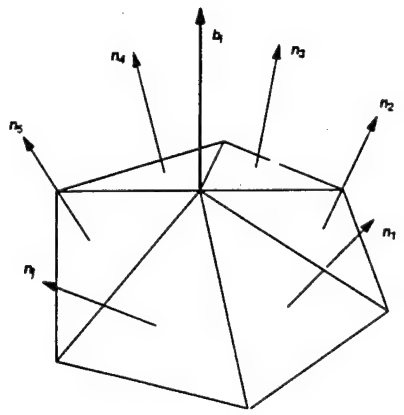


Figure 2.3: Node normals determined from surrounding face normals.

is re-ordered so that optimal element quality will be produced (concave first and convex last). Tetrahedral elements are created (only temporarily if mixed elements are desired) for a given new point by inflating all surrounding boundary faces as shown in Figure 2.4. Pentahedral elements can also be created using points generated on subsequent layers. Five and six node pentahedra are formed by combining tetrahedra as shown in Figure 2.5. With mixed element types, five-node pentahedra and tetrahedra are created only on the outer layer of the anisotropic region as shown in Figure 2.6. These elements are required so that the anisotropic region is bounded only by triangular faces. All of the elements in the combined grid have strict node, edge, and face matching to each other and to neighboring tetrahedral elements.

2.2.5 Isotropic Grid Generation

The AFLR grid generation procedure used in the present work is a combination of automatic point creation, advancing type ideal point placement, and connectivity optimization schemes. A

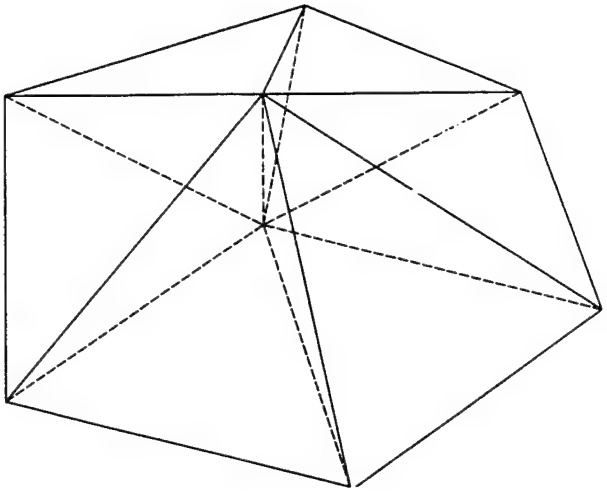


Figure 2.4: Creation of tetrahedral elements using advancing-normal point placement.

valid grid is maintained throughout the grid generation process. This provides a framework for implementing efficient local search operations using a simple data structure. It also provides a means for smoothly distributing the desired point spacing in the field using a point distribution function. This function is propagated through the field by interpolation from the boundary point spacing or by specified growth normal to the boundaries. Points are generated using advancing-front type point placement. The connectivity for new points is initially obtained by direct subdivision of the elements that contain them. Connectivity is then optimized by local-reconnection with a combined Delaunay and min-max (minimize the maximum angle) type criterion. The overall procedure is applied repetitively until a complete field grid is obtained. Complete details are presented in [11], [12].

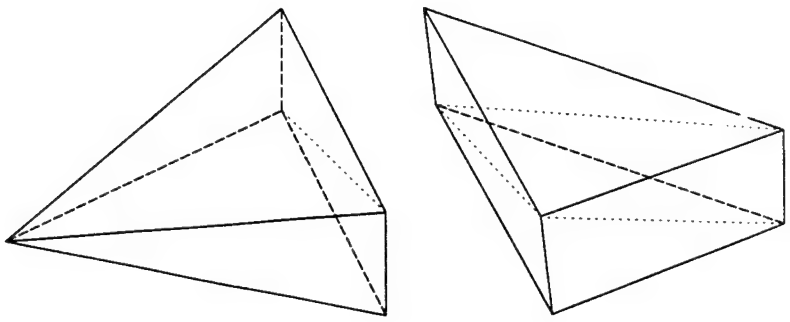


Figure 2.5: Pentahedral elements formed by combining tetrahedral elements.

2.3 Application Examples

Selected application examples are presented here to demonstrate the capabilities of the present procedure for generation of three-dimensional unstructured grids of mixed element types that are suitable for Reynolds-Averaged Navier-Stokes simulations. All geometry preparation and surface grid generation work was done using SolidMesh [13] with AFLR surface grid generation [12].

Grid quality distributions and statistics are presented for all examples in Figure 2.7. Element angle is used as the grid quality measure. The complete set of grid quality data consists of the six, eight, and nine dihedral angles for all tetrahedra, five-node pentahedra, and six-node pentahedra respectively. In Figure 2.7. the distribution plot is in 5 deg. increments. As shown, the distribution has peaks at 60 and 90 deg. from the six-node pentahedra and a peak near 70 deg. from the tetrahedra. The results for the examples presented are representative of

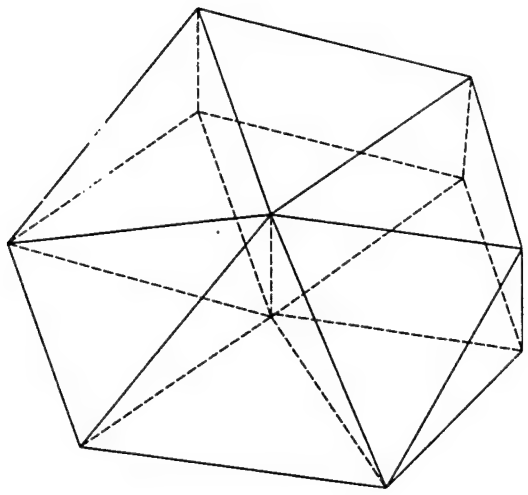


Figure 2.6: Use of pyramid and tetrahedral elements to transition between prismatic and tetrahedral regions.

those obtained for a variety of configurations. Typically, for isotropic elements in the grid, the maximum element angle is 160 deg. or less, the standard deviation is 17 deg. or less, and 99.5% or more of the element angles are between 30 and 120 deg. And, for anisotropic elements in the grid, the maximum element angle is 170 deg. or less and 99.5% or more of the element angles are between 30 and 135 deg. The minimum angle is usually dictated by the geometry. Convex or concave edges with an included angle less than 20 deg., such as a sharp trailing edge or the interior of a wedge, can produce larger maximum angles in the anisotropic region. The maximum anisotropic element angle can be controlled by specifying the maximum allowable angle (which eliminates generation of such elements) or by use of multiple normals at convex edges3.

CPU time required on a SUN Ultra 60 workstation is presented in Table 2.1 for each example. Computer routines for the three-dimensional grid generator are written in C with dynamic

memory that is automatically reallocated based upon actual requirements. All floating-point calculations are performed using 64 bit precision with 8 byte data. The CPU times reported are for one processor and include all I/O and generation of grid quality data. A boundary surface grid file is the input. The output includes a grid coordinate and connectivity file and a quality data file. Memory required is about 300 bytes per node generated. Requirements for memory and CPU time vary with the percentage of anisotropic elements, as those requirements for anisotropic generation are considerably less than those for isotropic generation.

User input required to generate a complete grid is minimal and includes specifying the point spacing at selected control points on the boundary curves for surface grid generation. Selection of options such as which boundaries to generate anisotropic elements from and initial normal spacing are the only required user input for volume grid generation. There are no user adjustable parameters that need to be changed from case to case. In all cases presented here, the initial normal spacing was set suitable for high Reynolds number viscous CFD analysis. Initial normal spacing was determined such that the first node adjacent to a viscous surface would have a y^+ value near 1.

Table 2.1: Number of nodes and elements generated and CPU time required for example cases.

Case	Boundary Faces	Nodes	Five-node Pentahedra	Six-node Pentahedra	Tetrahedra	CPU Time (min)
EET Wing Body	272,920	2,290,661	33,001	3,744,105	2,004,663	41.9
Space Shuttle Orbiter	152,810	1,102,869	10,269	1,709,948	1,181,232	16.6
Titan IV-A Exterior	337,596	2,980,493	25,370	5,113,326	1,962,705	59.4
Titan IV-A Interior	88,502	571,399	31,548	861,492	642,859	5.9

2.3.1 Energy Efficient Transport (EET)

A grid suitable for high Reynolds number viscous CFD analysis was generated for a high-lift wing/body configuration of the Energy Efficient Transport (EET). The surface grid on the underside of the wing is shown in Figure 2.8. Field cuts near the wing/body, wing/slat, and wing/flap regions are shown in Figure 2.9. Element size varies smoothly in the field and there is a smooth transition between the anisotropic and isotropic regions. In narrow regions between components, the number of anisotropic layers is reduced to produce high-quality elements between them. Also, the symmetry plane grid has been re-generated to match the interior anisotropic elements exactly. Grid quality distributions are shown in Figure 2.7. Number of boundary faces, nodes, elements and CPU time are presented in Table 2.1. An all tetrahedral element version of this grid was used by Sheng, et al [14] for incompressible flow simulation with an implicit multi-block flow solver.

2.3.2 NASA Space Shuttle Orbiter

A grid suitable for high Reynolds number viscous CFD analysis was generated for the NASA Space Shuttle Orbiter. The surface grid is shown in Figure 2.10. Field cuts near the rocket motor and inboard flap regions are shown in Figure 2.11. Element size varies smoothly in the field and there is a smooth transition between the anisotropic and isotropic regions. In narrow regions between components, the number of anisotropic layers is reduced to produce high-quality elements between them. Also, the symmetry plane grid has been re-generated to match the interior anisotropic elements exactly. Grid quality distributions are shown in Figure 2.7. Number of boundary faces, nodes, elements and CPU time are presented in Table 2.1.

2.3.3 Titan IV-A Launch Vehicle

A grid suitable for high Reynolds number viscous CFD analysis was generated for a Titan IV-A launch vehicle wind tunnel test model configuration. This configuration includes two strap-on solid rocket motors (SRM), thrust vector control (TVC), stage separation motor (SSM), interstage cavity, and wind tunnel test model sting. The overall configuration and surface grid

near the SRM, TVC and SSM regions are shown in Figure 2.12. Field cuts near the SRM, TVC, and SSM regions are shown in Figure 2.13. A field cut within the interstage cavity is shown in Figure 2.14. Element size varies smoothly in the field and there is a smooth transition between the anisotropic and isotropic regions. In narrow regions between components, the number of anisotropic layers is reduced to produce high-quality elements between them. Grid quality distributions are shown in Figure 2.7. Number of boundary faces, nodes, elements and CPU time are presented in Table 2.1. This configuration, and the others presented here, are representative of the high level of geometric complexity that can be handled routinely using the present approach.

2.4 Summary

A procedure has been presented for efficient generation of high-quality unstructured grids of mixed element types suitable for CFD simulation of high Reynolds number viscous flow fields. Layers of anisotropic elements are generated by advancing along prescribed normals from solid boundaries. The points are generated such that either pentahedral or tetrahedral elements with an implied connectivity can be be directly recovered. As points are generated they are temporarily attached to a volume triangulation of the boundary points. This triangulation allows efficient local search algorithms to be used when checking merging layers. The existing AFLR procedure is used to generate isotropic elements outside of the anisotropic region. Results were presented for a variety of applications. The results demonstrate that high-quality anisotropic unstructured grids can be efficiently and consistently generated for complex configurations.

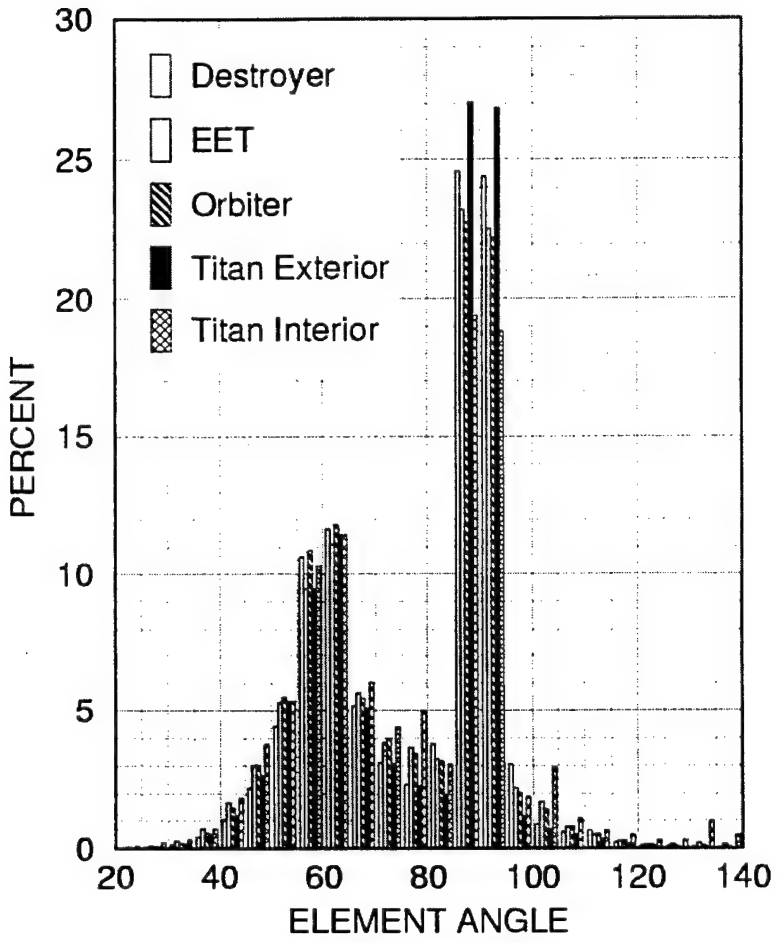


Figure 2.7: Volume element angle distributions.

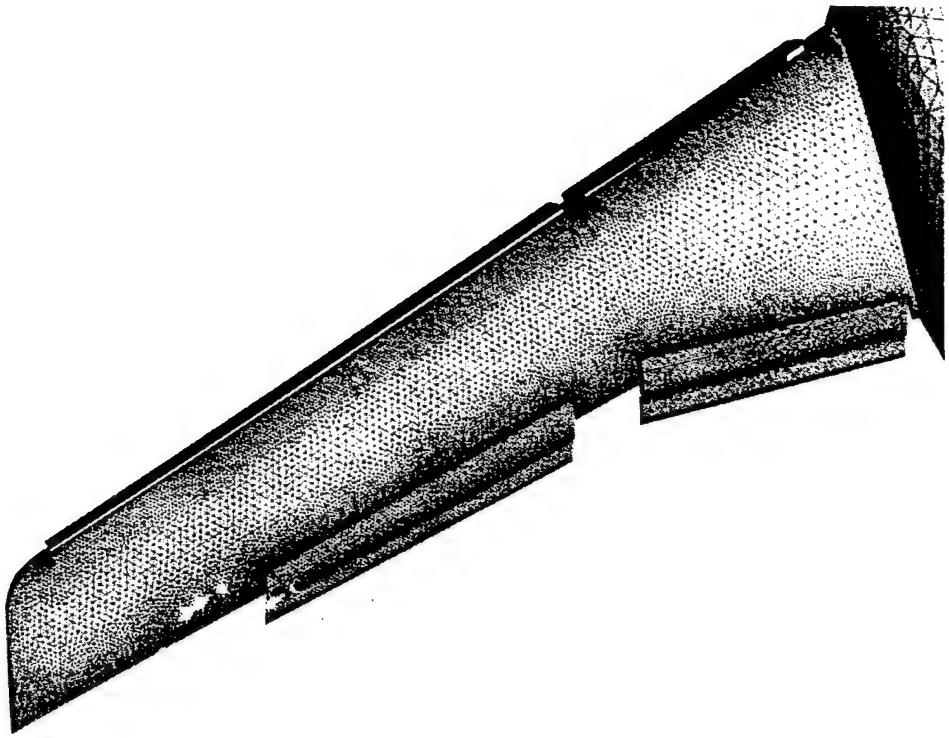
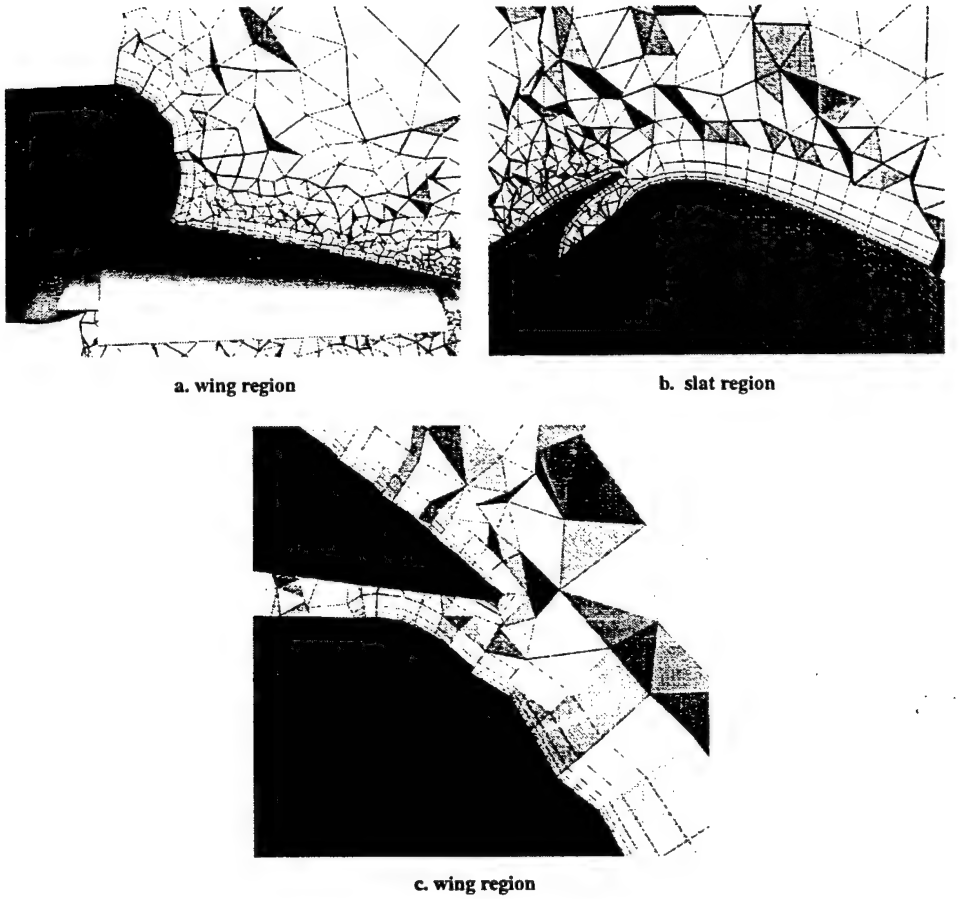


Figure 2.8: EET wing surface grid.



a. wing region

b. slat region

c. wing region

Figure 2.9: Field cuts for EET grid.

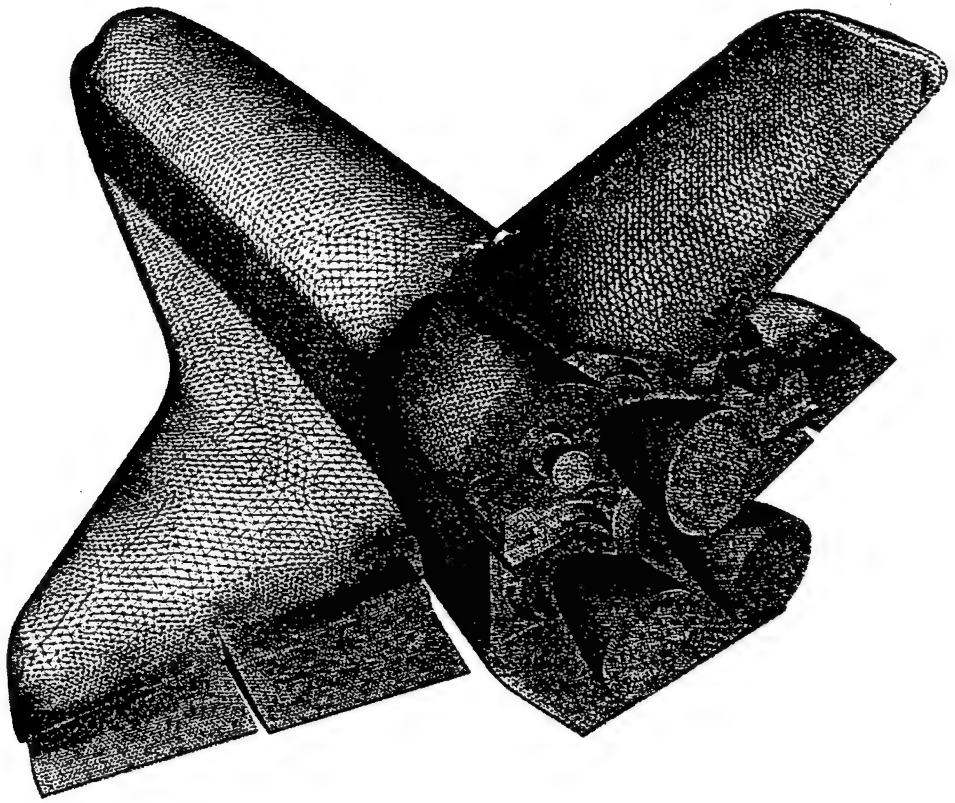
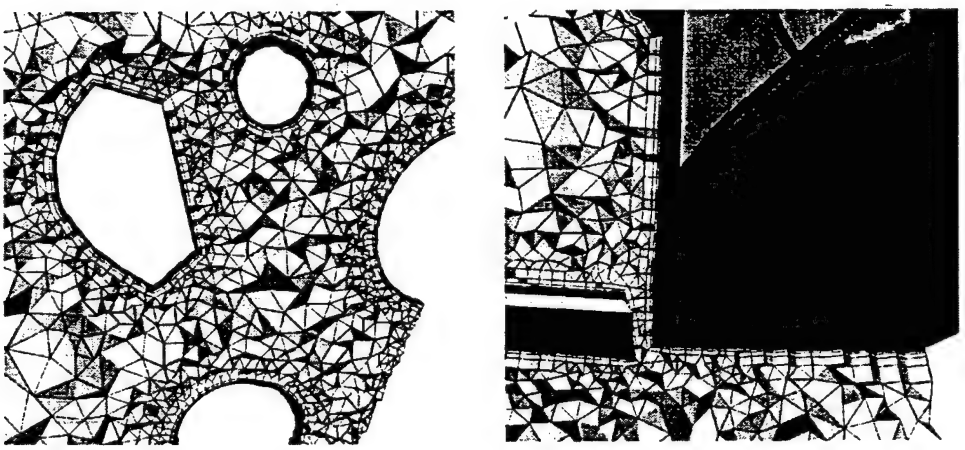


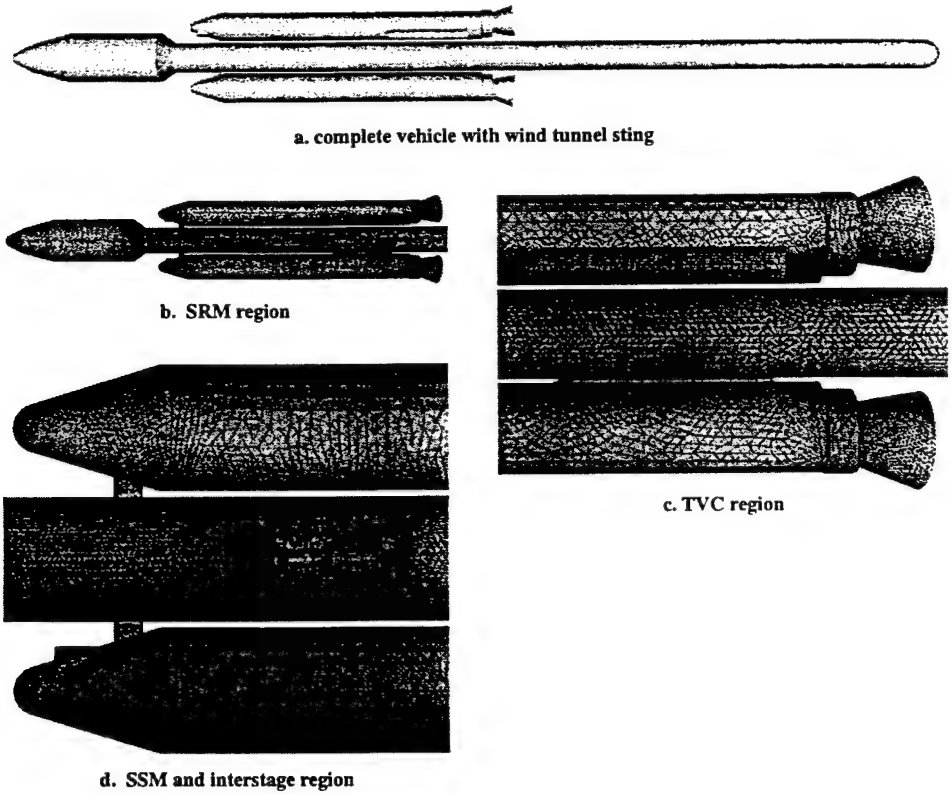
Figure 2.10: NASA space shuttle orbiter surface grid.



a. rocket motor region

b. inboard flap region

Figure 2.11: Field cuts for NASA space shuttle orbiter grid.



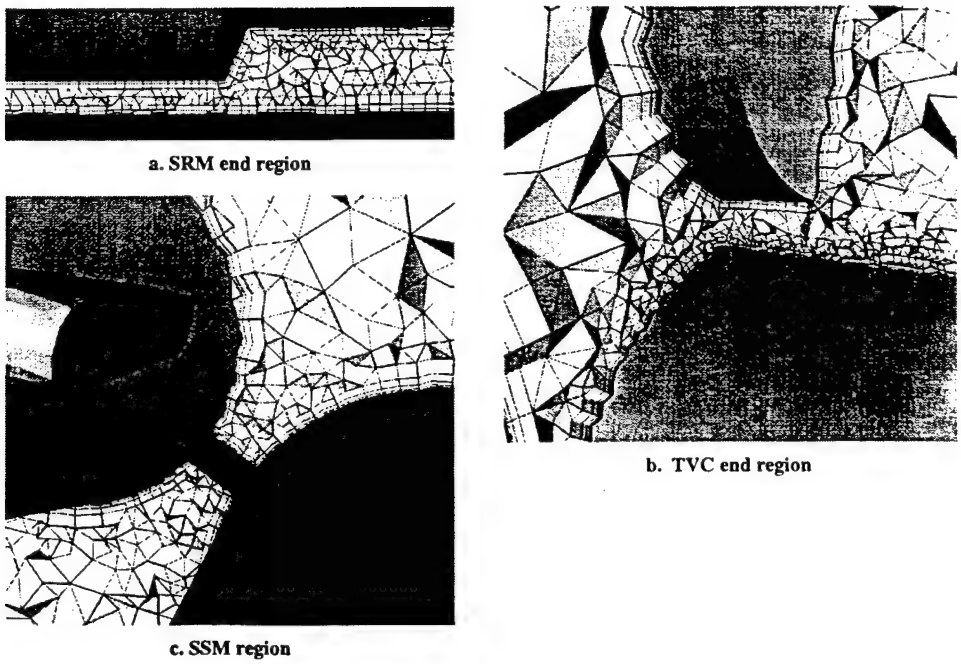
a. complete vehicle with wind tunnel sting

b. SRM region

c. TVC region

d. SSM and interstage region

Figure 2.12: Titan IV-A launch vehicle surface and grid.



a. SRM end region

b. TVC end region

c. SSM region

Figure 2.13: Field cuts for Titan IV-A launch vehicle external grid.

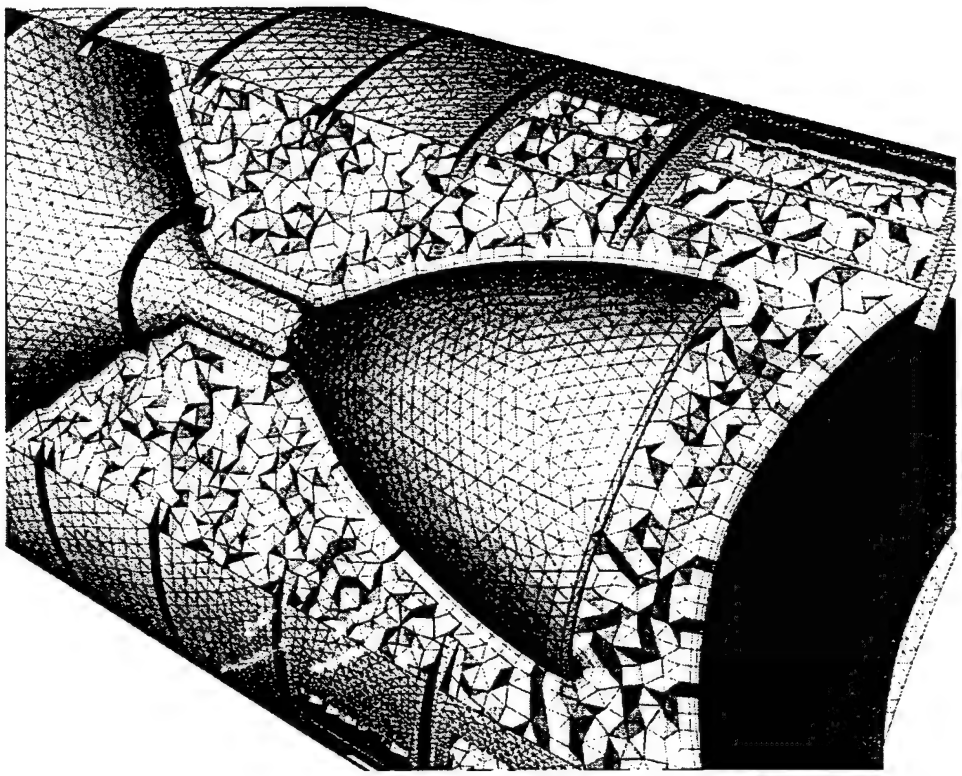


Figure 2.14: Field cut and surface grid for Titan IV-A interstage cavity.

CHAPTER III

RELATIVE BODY MOTION

3.1 Introduction

Unsteady simulations for moving geometries typically require a body conforming grid at all times. If the bodies in the flow field undergo arbitrary movement, a fixed grid will lead to badly distorted elements which will result in convergence difficulties and poor quality results. Remeshing must be carried out in order to have a body conforming grid. One option is to do a global regeneration which is an expensive process and can degrade the accuracy due to accumulation of interpolation errors.

Unsteady flows with moving bodies are often handled using adaptive remeshing [15, 16] of regions undergoing rapid changes. It has been observed that frequent adaptations may lead to poor numerical results and also can result in loss of essential physical features of the flow [17]. The poor performance is due to the interpolation of data from one grid to another. In order to overcome this loss of information and to increase the efficiency, local remeshing [17] can be carried out in the vicinity of highly distorted elements. A typical simulation may require several regenerations. In order to carry out realistic computations, a fast remeshing capability is required.

Identification of the region of grid deformation is an important task in the dynamic grid generation procedure. One approach is the adaptive window procedure presented by Singh et al [18]. Windows are created by specifying a normal distance from the body of interest. The entire domain is searched to locate the points that fall within the window and are flagged as window points, which is quite expensive. The window points are considered as a spring network and are allowed to adapt to the body movement. Tension spring analogy is a popular technique that has been used to solve moving body problems [18], [19] and has been proven

successful for problems with small scale deformation [19]. For large deformation, the spring stiffness is critical and crossing of grid lines may be encountered since the connectivity has to be maintained at all times. Finite element methods have also been used to solve the moving body problems [17, 20]. An arbitrary Lagrangian-Eulerian(ALE) formulation has been used for solving transient problems with large scale deformations [20] in which the coordinates can move in an arbitrary way.

Most of the previously described methods were tested on inviscid grids. Many practical situations involve solution of viscous flows, which requires a high aspect ratio grid close to the body. In this case, the problem becomes more severe since the deformation of the high aspect ratio elements will result in a poor quality grid with slight movement of the bodies. Hence, we need a general method that can handle both viscous and inviscid grids with arbitrary motions. Another real challenge for the dynamic mesh algorithm is in its ability to handle the relative motion of the bodies in close proximity. In this case, badly distorted elements are encountered with minimal deformation. Frequent remeshing is required to maintain grid quality.

In the present study a general dynamic unstructured grid algorithm is presented, which can handle arbitrary motion. An efficient procedure for the identification of the window region is developed. The problem of handling the relative motion of bodies in close proximity has been addressed by developing a local marching procedure. Results are presented for mixed element type grids to demonstrate the efficiency of the present algorithm. Topological changes are not considered in the present research.

3.2 Grid Generation

Grid generation for a given geometry is carried out using the Advancing-Front/Local-Reconnection(AFLR) [21, 12, 22] method. The AFLR procedure uses a combination of automatic point creation and advancing type ideal point placement. A point distribution function is assigned to each boundary point based on the local spacing. The growth rate normal to boundary is also specified for each boundary point. The point distribution function is propagated through the field by interpolation. Points are generated using advancing-front type

point placement for isotropic elements [21], and advancing-normal type point placement for high-aspect ratio elements [22]. Initial connectivity is obtained using direct subdivision. A valid grid is maintained throughout the grid generation process. The grid connectivity is then optimized using iterative local-reconnection process subject to a quality criterion. A combined Delaunay/min-max type criterion is used. The overall procedure is repeated until the entire field grid is generated with desired point spacings.

For complex geometries a mixed element type grid is generated with pseudo-structured elements in the regions where the geometry is smooth. To improve the efficiency of the flow solver unstructured tetrahedral elements are used elsewhere. The advancing normal type point placement used for the generation of the high-aspect ratio elements inside the boundary layer leads to structured type elements. The elements inside the boundary layer are combined to form mixed(pentahedral and tetrahedral) elements [22].

The dynamic grid generation process can involve a number of regenerations. Therefore, an efficient grid generation procedure is extremely important for a moving body simulation. The AFLR procedure has a demonstrated ability to generate high quality grids about geometrically complex configurations for a variety of applications [12, 22]. Also this procedure is highly efficient and robust, thus providing a direct contribution to the efficiency of the overall dynamic grid generation algorithm.

3.3 Window(Deforming Region) Identification

Identification of windows corresponding to the moving bodies plays a significant role in the dynamic grid generation process. The ability of the procedure to identify this region, has an immediate consequence in the overall efficiency, since the windows will have to be recreated many times during the simulation. The first step in the present dynamic grid generation algorithm is to form the protected layers corresponding to all the stationary surfaces in a given grid. This is done in order to avoid the intersection of body surfaces and to preserve the boundary layer of the stationary bodies. Rigid layers are formed corresponding to each moving body, that are allowed to move with the body. If a moving body has a viscous boundary condition then the

entire boundary layer is allowed to move with the body in a rigid fashion. This prevents the distortion of the high aspect ratio boundary layer elements and preserves a high quality grid. Windows are identified outside the rigid layers and the nodes that fall inside this window region are allowed to adapt to the body movement.

3.3.1 Marching Procedure

The protected layers, rigid layers and windows are formed by using a marching procedure. The list of elements surrounding a node is built using the grid connectivity information. The nodes that appear on the moving body surface are tagged as surface nodes. Marching is carried out from the surface using the list of elements surrounding a node. The nodes of elements that do not belong to the current layer are tagged as nodes corresponding to the next layer. Marching is continued until a specified number of layers are identified. This procedure is highly efficient, since marching is carried out locally from only those nodes that are tagged as belonging to the surface of interest. In the case of mixed element type grids the window region contains only the tetrahedral elements as all the mixed elements occur inside the boundary layer. The end of the rigid layer and that of the window region define the boundaries of the deforming region. These boundaries form a valid surface grid used when the deforming region undergoes regeneration based on the quality criterion. The outer boundary of the window forms the interface between the deforming and non-deforming regions.

3.3.2 Local Marching

Modeling of practical geometries often includes bodies in close proximity with narrow gaps e.g. launch vehicle geometries with strap-on boosters. The number of layers generated between the bodies that are very close is hence restricted by the size of the gap. Uniform marching in such gaps will result in either the intersection of the boundary layer or the surface of the neighboring body. Therefore a local marching procedure has been developed. Marching is stopped locally on any region if a node comes in contact with the nodes that are already marked. This allows the layers to grow only in those regions where elements are available for marching. Intersection of

the window regions is another problem that is encountered when moving bodies are close to each other. This problem is treated by identifying a common window for all bodies whose windows intersect. The common window is used until the bodies move sufficiently far to establish their independent windows.

3.4 Grid motion and Remeshing

Grid motion has to be carried out in order to follow the moving boundary. The motion is implemented in two steps. The first step requires the rigid layers attached to the body to move with the body to which they are associated. This helps in keeping the grid undistorted close to the surface. Deformation is carried out in the grid region inside the window in such a way that element distortion is minimized. This is achieved by calculating weights associated with the nodes. All the nodes corresponding to moving surfaces and the rigid layers have a weight of one and the nodes in the non-deforming region and the nodes corresponding to all other stationary surfaces will have a weight of zero. The weights of the nodes inside the window region varies smoothly from one to zero as shown in Figure 1.

After each deformation the grid quality of the window region is checked by calculating the dihedral element angle. A low quality element will have an angle close to 180 degrees. Elements with an angle of 170 degrees are typically acceptable. In cases when the bodies are in close proximity, the maximum angle of the undeformed window region is used for the quality criterion. In addition to the angle criterion, the element volume is checked. If a given element has a volume which differs from its original volume by more than a factor of three then the quality criterion is not considered satisfied. A volume ratio of three is considered acceptable during the deformations.

If the quality criterion are not satisfied, a local regeneration of the window region is carried out. Surface reconnection is not allowed during this regeneration process since that would effect the connectivity of the non-deforming portion of the grid. The regenerated grid is then put back to form a new grid. The new grid is checked to make sure that there are no zero volume elements.

Deformation is continued after recomputing the weights and windows. When two bodies are in close proximity, this procedure could result in elements with zero volume on the interface between the rigid layers and the regenerated grid. This occurs due to the restriction placed on the surface grid that no reconnection is allowed. Under such conditions a second level of regeneration is carried out by remeshing the grid from the body surface to the end of the window region. If the moving body has a viscous boundary condition, the boundary layer is regenerated. The overall grid quality is maintained. In some cases, it may also be possible to eliminate the zero volume elements by reconnecting the elements in the fixed region. Results presented in the next section shows the efficiency of the present algorithm, which is mainly due to the fact that the regenerations are localized.

3.5 Results

To validate the present dynamic grid generation algorithm a launch vehicle test case with three boosters is considered This configuration consists of bodies in close proximity which poses a real challenge for the dynamic grid generation process. Details of the grid generation of this geometry can be found in Ref [12]. A viscous grid is generated for the given configuration using the AFLR technique with mixed element types inside the boundary layer. The initial grid consists of 390,332 nodes with 628,827 tetrahedron, 4,439 five-node pentahedron(pyramids) and 541,903 six-node pentahedron(prisms). Rigid layer and window region statistics corresponding to the moving booster are shown in Table 1. The CPU time reported is the time to generate the initial maps for the rigid and window regions.

Table 3.1: Rigid layer and window region statistics

	No. of layers	% nodes	% tets	% pyramids	% prisms	CPU (sec)
Rigid layers	17	12.022	0.136	20.20	15.922	2.41
Window region	5	1.876	10.449	0	0	2.05
	10	4.739	22.327	0	0	3.62

The computations are carried out on one processor of a Sun HPC 10000. In order to bring out the effect of the number of layers in the window region on the quality of the deformed grid and the efficiency of the algorithm, a detailed study is carried out by considering both five and ten layers in the window region. The distribution of the weights in the window region for the initial grid with five and ten layers is shown in Figures 1a and 1b and the final position in Figures 1c and 1d. It can be seen from the figures that the small gap between the main body and booster restricts the growth of the window region and therefore a local marching procedure is used for the window identification.

The dynamic grid generation is carried out by allowing one of the boosters to follow a translational motion. A ΔX of 0.025 is used for the deformation. The overall length of the moving strap-on booster is 47.6. 500 deformation steps are carried out. It is observed that, with the increase in distance between the main body and the booster, there is an increase in the number of elements and nodes in the window region. After the booster has moved far enough to establish a fully developed window region, a decrease in the number of elements and nodes in the window region is noted. The number of elements in the window region ranged from 65646 to 82728 in the five layer case. Local regeneration of the window region is carried out when the the quality criterion is violated. The details of the simulation are presented in Table 2. The total CPU time listed is for all grid work during the simulation, including grid motion, remeshing, regeneration of layer maps and calculation of weights.

Table 3.2: Remeshing data for strap-on booster separation simulation

No. of layers	No. of steps	Total CPU (sec)	AFLR CPU	Other CPU	No. of local(AFLR) remeshing(level 1)
5	500	3268.65	281.58	2987.07	27
10	500	2645.9	230.79	2415.11	7

Other than the AFLR code for mesh generation, the present code is not optimized. With a fully optimized code, a good improvement in the efficiency of the algorithm is anticipated. Presently, several of the required maps are generated globally rather than locally for convenience. Field

cuts of the deformed grid with five and ten layers in the window region are shown in Figures 2 and 3. Figure 2 shows that the moving booster leaves a trail near the tip as it moves. With ten layers in the window region the trail is cleared and it can be seen from Figure 3 that the deformed grid is of good quality. Figure 4 shows the field cut of the final grid with two moving boosters and five layers in the window region.

The results of the present study show that an increase in the number of layers in the window region results in a reduction in the number of local regenerations, and a corresponding increase in efficiency. Accuracy of the overall flow simulation algorithm should also increase due to reduced interpolation errors.

3.6 Conclusions

A general dynamic grid algorithm capable of handling arbitrary relative motion of multiple bodies is presented. A robust procedure for identifying the windows is developed and the treatment of bodies in close proximity is addressed by using a local marching procedure.

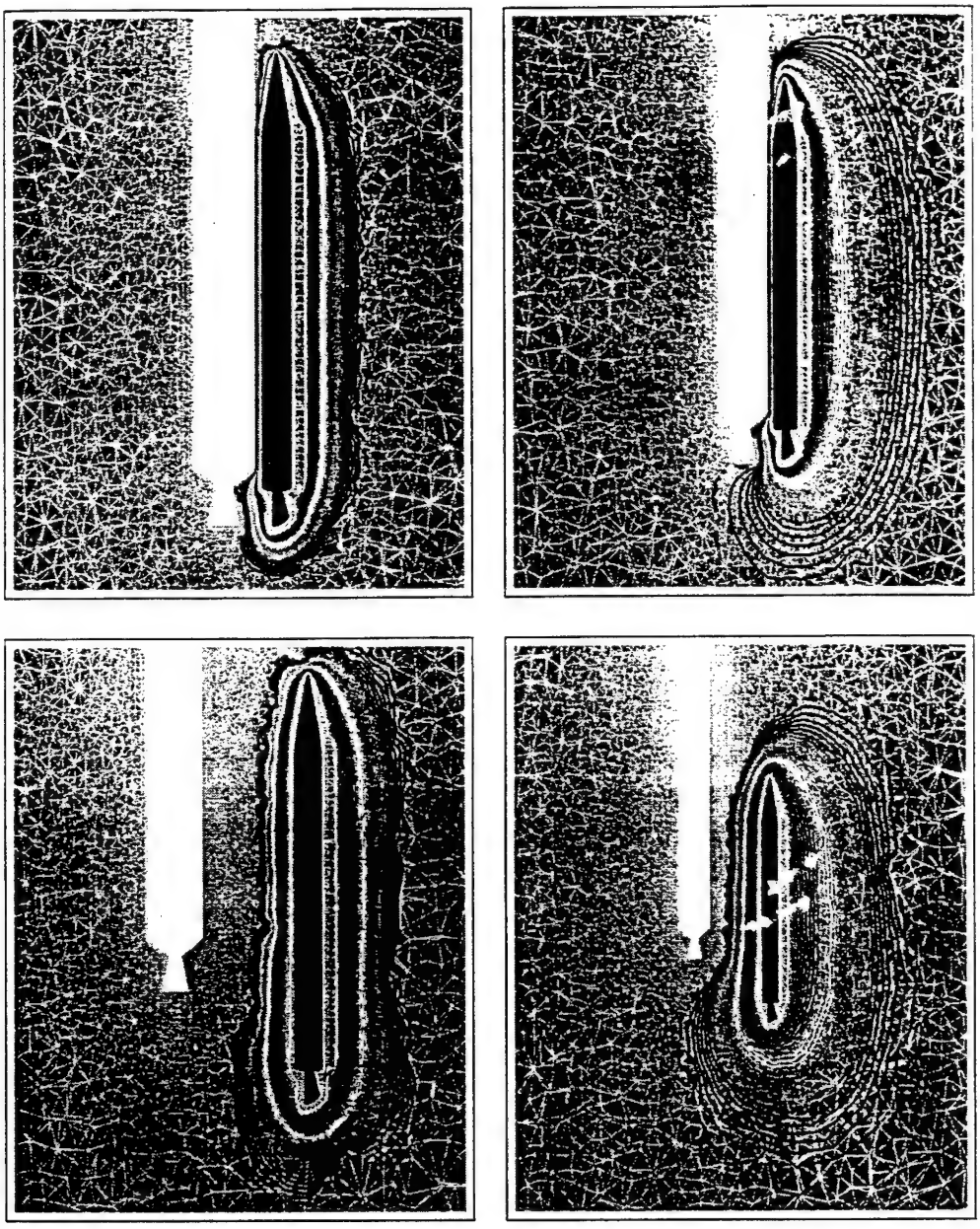


Figure 3.1: Window region a. 5 layers initial, b. 10 layers initial, c. 5 layers final, d. 10 layers final

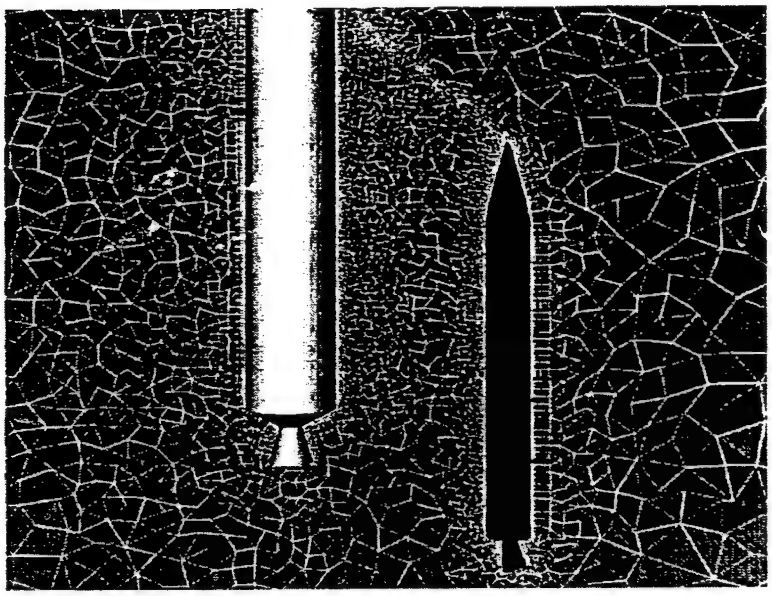


Figure 3.2: Field cut after 500 steps, with 5 layers in window region

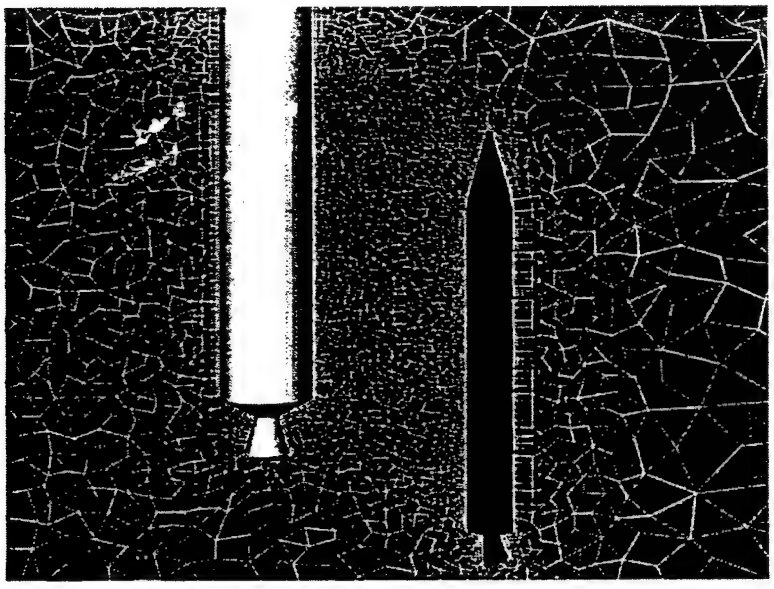


Figure 3.3: Field cut after 500 steps, with 10 layers in window region

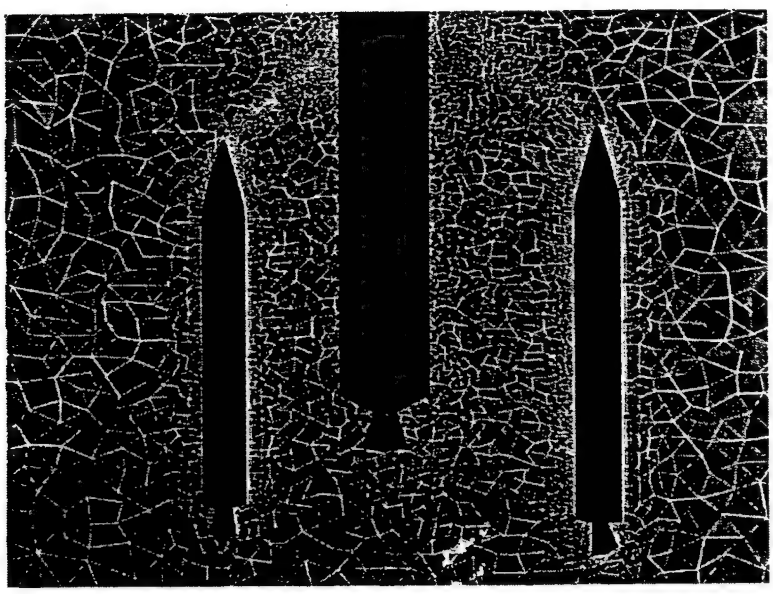


Figure 3.4: Field cut after 500 steps, with 10 layers in window region

CHAPTER IV

SIX DEGREE OF FREEDOM (6DOF) MODEL

Prediction of the trajectory of the moving body requires the coupling of the fluid dynamic equations and the solid body equations. The application of Newton's second law to moving bodies results in the six-degree-of-freedom(6DOF) equations. Details of the derivation of the equations can be found in Stevens et al [23]. Also, a detailed discussion of the equations of rigid body motion can be found in [24].

Two sets of coordinate systems are used in this model. The body fixed system is rigidly attached to the moving body with the Center of Gravity(CG) as its origin. There is an inertial coordinate system to which the position and orientation of the body is referenced. B(t) is the rotation matrix that takes vectors from the inertial to the body coordinate system.

The state variables of the model consists of three components of each position vector $\vec{p} = (x, y, z)^T$, the translational velocity coordinates $\vec{v}_B = (u, u, w)^T$ and the angular velocity coordinates $\vec{\omega}_B = (P, Q, R)^T$. A differential equation is needed for time-varying transformation matrix B, which will lead to four additional equations for the attitude in terms of the quaternions $\vec{q} = (q_0, q_1, q_2, q_3)^T$.

One set of state equations is obtained by writing $\dot{\vec{p}}$ in terms of its components

$$\dot{\vec{p}} = B^T \vec{v}_B + \vec{\omega}_E \times \vec{p}$$

or

$$\dot{\vec{p}} = B^T \vec{v}_B + \Omega_E \vec{p} \tag{4.1}$$

where $\vec{\omega}_E$ is the absolute angular velocity of Earth's rotation. The symbol Ω is used to denote the cross-product matrix corresponding to the operation $(\vec{\omega}\times)$.

Newton's second law, applied to the translational motion, relates force to rate of change of linear momentum which is given by

$$\vec{F}_B + Bm\vec{g} = m\frac{d}{dt_I}[\vec{v}_B + B(\vec{\omega}_E \times p)] \tag{4.2}$$

where $\vec{F}_B$ is the force in the body coordinate system, $Bm\vec{g}$ is the gravity force rotated into body frame by rotation matrix B, and $\frac{d}{dt_I}(m\vec{v}_B)$ is the time rate of change of linear momentum of the body with respect to the inertial coordinate system. Expansion of this last term results in

$$\frac{d}{dt_I}[m\vec{v}_B + B(\vec{\omega}_E \times p)] = (\dot{\vec{v}}_B + \vec{\omega}_B \times \vec{v}_B) + B(\vec{\omega}_E \times \dot{\vec{p}}) \tag{4.3}$$

Rearranging the Eq.(4.2), using Eqs.(4.1,4.3) the state equation for the translational velocity becomes

$$\dot{\vec{v}}_B = \frac{\vec{F}_B}{m} - (\vec{\omega}_B + B\vec{\omega}_E) \times \vec{v}_B + B[\vec{g} - \vec{\omega}_E \times (\vec{\omega}_E \times \vec{p})] \tag{4.4}$$

The angular accelerations are obtained by applying Newton's second law to the rate of change of angular momentum of the body. The angular equation of motion is given by

$$\vec{T}_B = \frac{d}{dt_I}(\vec{H}_B) \tag{4.5}$$

$\vec{T}_B$ is the net torque acting about the body CG, $\vec{H}_B$ is the angular momentum of the rigid body and the time derivative is with respect to the inertial coordinate system. Expansion of the time derivative term results in

$$\frac{d}{dt_I}(\vec{H}_B) = \dot{\vec{H}}_B + \vec{\omega}_B \times \vec{H}_B \tag{4.6}$$

The angular momentum is given by

$$\vec{H}_B = J\vec{\omega}_B \tag{4.7}$$

where J is the matrix of moment of inertia

$$J = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix}$$

The entries of the inertia matrix are computed as follows

Moment of inertia about the x-axis is given by

$$J_{xx} = \int (y^2 + z^2) dm \tag{4.8}$$

Cross-product of inertia is given by

$$J_{xy} \equiv J_{yx} = \int xy dm \tag{4.9}$$

The state equation for the rotational motion is given by

$$\dot{\vec{\omega}}_B = -J^{-1}(\vec{\omega}_B \times (J\vec{\omega}_B)) + J^{-1}\vec{T}_B \tag{4.10}$$

A four-variable attitude propagation in terms of the quaternions is considered in the present work to determine the orientation of the body. The three-variable attitude propagation in terms of Euler angles has some disadvantages. When the pitch angle θ reaches ± 90 degrees, a division by zero is encountered. Also, Euler angles may integrate up to values outside the the normal range of the pitch, roll and yaw angles [23]. Therefore it is difficult to determine the attitude

uniquely. Four-variable attitude propagation overcomes this problem. The differential equation for the quaternion parameters is given by

$$\dot{\vec{q}} = -\frac{1}{2}\Omega_q \vec{q} \tag{4.11}$$

4.1 The Round-Earth Equations

For a complete state model the relevant state equations are assembled in a matrix form (Ref. [23]) as follows

$$\begin{bmatrix} \dot{\vec{p}} \\ \dot{\vec{v}}_B \\ \dot{\vec{\omega}}_B \\ \dot{\vec{q}} \end{bmatrix} = \begin{bmatrix} \Omega_E & B^T & 0 & 0 \\ -B\Omega_E^2 & -(\Omega_B + B\Omega_E) & 0 & 0 \\ 0 & 0 & -J^{-1}\Omega_B J & 0 \\ 0 & 0 & 0 & -\frac{1}{2}\Omega_q \end{bmatrix} \begin{bmatrix} \vec{p} \\ \vec{v} \\ \vec{\omega} \\ \vec{q} \end{bmatrix} + \begin{bmatrix} 0 \\ B\vec{g} + \frac{\vec{F}_B}{m} \\ J^{-1}T_B \\ 0 \end{bmatrix} \tag{4.12}$$

The state vector $\vec{X}^T = \left[\vec{p}^T, \vec{v}_B^T, \vec{\omega}_B^T, \vec{q}^T\right]$ contains 13 elements and it completely determines the position of the body at any given instant.

The coefficient matrix consists of submatrices

$$\Omega_E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -\omega_x \\ 0 & \omega_x & 0 \end{bmatrix}; \quad \Omega_B = \begin{bmatrix} 0 & -R & Q \\ R & 0 & -P \\ -Q & P & 0 \end{bmatrix}; \quad \Omega_q =$$

$$\begin{bmatrix} 0 & P & Q & R \\ -P & 0 & -R & Q \\ -Q & R & 0 & -P \\ -R & -Q & P & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

The rotation matrix B is orthogonal. Hence $B^{-1} = B^T$. The forces and moments on the right hand side are functions of $\vec{v}_B$ and $\vec{\omega}_B$. The Eq.4.12 is nonlinear since the coefficient matrix B, Ω_B and Ω_q are functions of the state variables. The system of equations is numerically integrated in time using a four stage Runge-Kutta method. Earth's rotation rate is not considered in the present work.

4.2 Results

In order to validate the present code a test case with no forces is considered since an exact solution can be obtained by integrating the state equations. When there are no forces acting on the body, the rotational equation(Ref. [25]) reduces to

$$A\dot{\omega}_1 = (B - C)\omega_2\omega_3 \tag{4.13}$$

$$B\dot{\omega}_2 = (C - A)\omega_3\omega_1 \tag{4.14}$$

$$C\dot{\omega}_3 = (A - B)\omega_1\omega_2 \tag{4.15}$$

where A,B,C are the principal moments.

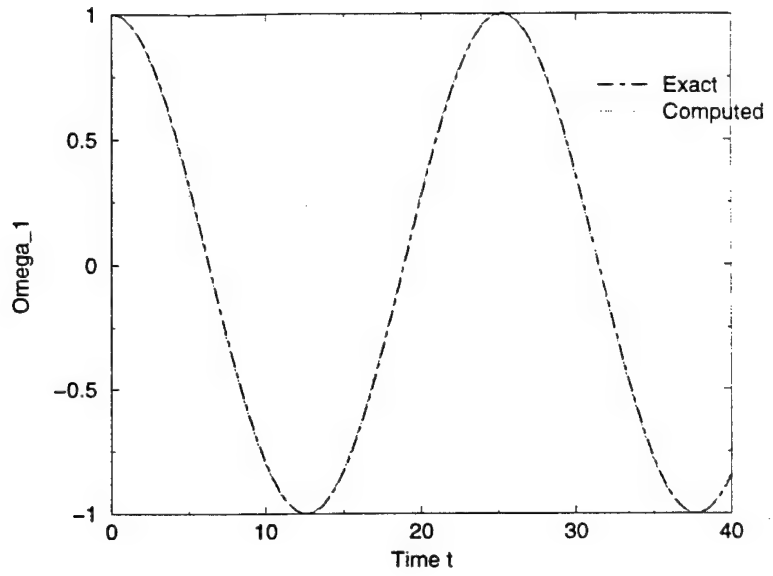


Figure 4.1: Comparison of the exact and computed solution for ω_1

Consider the case when two principal moments of inertia are equal A=B; then, $\dot{\omega_3} = 0$ or $\omega_3 = h$, a constant. The initial conditions at t=0 are $\omega_1 = a, \omega_2 = 0$ and $\omega_3 = c = h$. The exact solution (Ref. [25]) is obtained by integrating the Eq.(4.13,4.14) and is given by

$$\omega_1 = acos(\lambda t) \tag{4.16}$$

$$\omega_2 = sin(\lambda t) \tag{4.17}$$

$$\omega_3 = constant = h \tag{4.18}$$

where $\lambda = ac$.

800 time steps are used to reach the final time t=40. It can be seen from Figures 1 and 2 that there there is a good agreement between the exact and computed solutions.

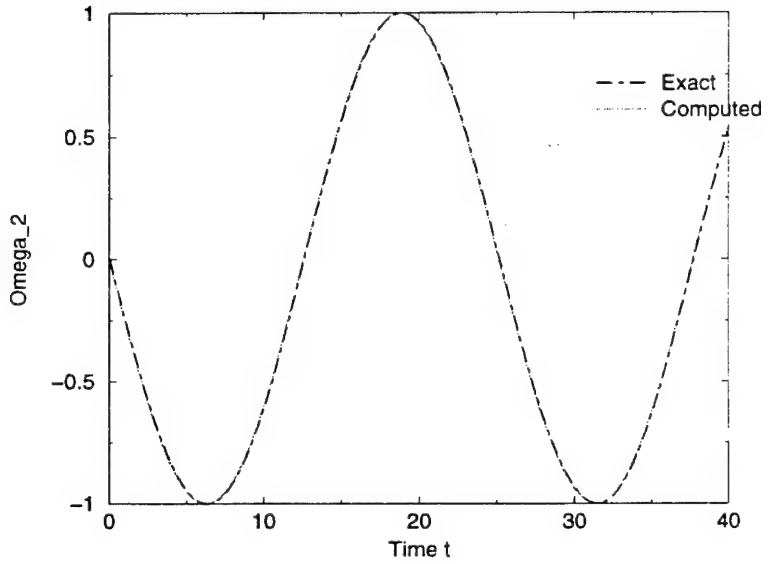


Figure 4.2: Comparison of the exact and computed solution for ω_2

CHAPTER V

COMPUTATIONAL METHODOLOGY

This chapter is intended to outline each of the techniques used to construct the present Navier-Stokes unstructured solution algorithm.

5.1 Governing Equations

The solution algorithm discussed in the present work is capable of handling both the Euler and Navier-Stokes equations. Thus, the equations given here are the full 3D Navier-Stokes equations, with the understanding that the the Euler equations do not contain the viscous terms. The unsteady three-dimensional compressible Reynolds-averaged Navier-Stokes equations are presented here in Cartesian coordinates and in conservative form. The nondimensionalized equations can be written in integral form as:

$$\frac{\partial}{\partial t}\int_{\Omega} Q d\mathcal{V} + \int_{\partial\Omega} \vec{F}\cdot\hat{\vec{n}} dA = \frac{M_\infty}{Re}\int_{\partial\Omega} \vec{G}\cdot\hat{\vec{n}} dA \tag{5.1}$$

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{bmatrix}$$

$$\vec{F} = \begin{bmatrix} \rho u \\ \rho u\,(u - V_x) + p \\ \rho v\,(u - V_x) \\ \rho w\,(u - V_x) \\ e\,(u - V_x) + pu \end{bmatrix} \hat{\imath} + \begin{bmatrix} \rho v \\ \rho u\,(v - V_y) \\ \rho v\,(v - V_y) + p \\ \rho w\,(v - V_y) \\ e\,(V - V_y) + pv \end{bmatrix} \hat{\jmath} + \begin{bmatrix} \rho w \\ \rho u\,(w - V_z) \\ \rho v\,(w - V_z) \\ \rho w\,(w - V_z) + p \\ e\,(w - V_z) + pw \end{bmatrix} \hat{k}$$

$$\vec{G} = \begin{bmatrix} 0 \\ \tau_{xx} \\ \tau_{yx} \\ \tau_{zx} \\ u\tau_{xx} + v\tau_{yx} + w\tau_{zx} - q_x \end{bmatrix} \hat{i} + \begin{bmatrix} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{zy} \\ u\tau_{xy} + v\tau_{yy} + w\tau_{zy} - q_y \end{bmatrix} \hat{j} + \begin{bmatrix} 0 \\ \tau_{xz} \\ \tau_{yz} \\ \tau_{zz} \\ u\tau_{xz} + v\tau_{yz} + w\tau_{zz} - q_z \end{bmatrix} \hat{k}$$

where the shear stresses are given as

$$\tau_{xx} = (\mu + \mu_t)\left[2\frac{\partial u}{\partial x} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right] \tag{5.2}$$

$$\tau_{yy} = (\mu + \mu_t)\left[2\frac{\partial v}{\partial y} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right] \tag{5.3}$$

$$\tau_{zz} = (\mu + \mu_t)\left[2\frac{\partial w}{\partial z} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right] \tag{5.4}$$

$$\tau_{xy} = \tau_{yx} = (\mu + \mu_t)\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \tag{5.5}$$

$$\tau_{xz} = \tau_{zx} = (\mu + \mu_t)\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \tag{5.6}$$

$$\tau_{yz} = \tau_{zy} = (\mu + \mu_t)\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \tag{5.7}$$

and the heat flux is defined as

$$\nabla q = -\frac{1}{\gamma - 1}\left(\frac{\mu}{Pr} + \frac{\mu_t}{Pr_t}\right)\nabla T \tag{5.8}$$

where T is the temperature, μ and μ_t are the laminar and turbulent viscosities and Pr and Pr_t are the laminar and turbulent Prandtl numbers respectively. The above equations are closed by

the equation of state for an ideal gas, i.e.,

$$e = \frac{p}{\gamma - 1} + \frac{1}{2}\left(u^2 + v^2 + w^2\right) \tag{5.9}$$

where γ is the ratio of specific heats and is taken to be 1.4. The above equations were nondimensionalized with respect to the freestream speed of sound (a_∞), freestream density (ρ_∞), characteristic length scale (L), and the freestream viscosity (μ_∞). Thus, $Re = \rho_\infty U_\infty L/\mu_\infty$ and M_∞ is the freestream Mach number. The nondimensional pressure is defined as $p = \frac{p^*}{(\rho_\infty U_\infty^2)}$, where p^* is the local static pressure. Obviously, for laminar flow, $\mu_t = 0$.

5.2 Spatial Discretization

Before discretization takes place, a control volume must be defined. The approach used in the present work is to define a control volume surrounding each vertex; thus, the solution technique is referred to as a vertex-centered (or node-centered) scheme.

All solution variables are stored associated with control volumes. To define specifically the control volume boundaries, the median dual is used; this dual construction consists of connecting the centroid of each incident element to the midpoint of each incident edge. The non overlapping volumes formed by this procedure are defined to be the control volumes over which flux balances are performed. The definition of the median dual in two dimensions in shown in Figure 5.1. Note that in three dimensions, element centroids are connected to face centroids as well as edge midpoints; this also forms a closed control volume around a vertex.

The governing equations are discretized using a finite volume technique; thus, the surface integrals in Equation 5.1 are approximated by a quadrature over the surface of the control volume of interest. So, the numerical discretization of the spatial terms associated with the control volume surrounding vertex 0 results in

$$\frac{\partial q_0}{\partial t} + \Re_0 = 0 \tag{5.10}$$

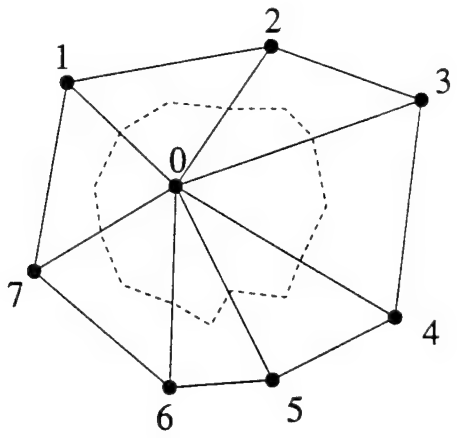


Figure 5.1: Control volumes are defined as median duals surrounding each vertex.

where the spatial residual $\Re$ contains all contributions from the discrete approximation to the inviscid and viscous terms ($\Re = \Re_{inv} + \Re_{vis}$). Also, the quantity q is defined as $q = \int_\Omega Q d\mathcal{V}$.

5.2.1 Inviscid Terms

Now, the integral for the inviscid terms must be approximated by a discrete sum; quadrature points are chosen as the midpoint of each edge incident to the vertex. Doing this, the flux vector is replaced by a suitable approximation $\vec{\Phi}$ (termed the "numerical flux vector"), therefore arriving at

$$\Re_{0,inv} = \sum_{i \in \mathcal{N}(0)} \vec{\Phi}_{0i} \cdot \vec{n}_{0i} \tag{5.11}$$

The numerical flux is calculated using the Roe scheme, which solves a one dimensional approximate Riemann problem given the two solution states on each side of the control volume face:

$$\Phi = \frac{1}{2}\left(F(Q_L) + F(Q_R)\right) - \frac{1}{2}\tilde{A}\left(Q_R, Q_L\right)\left(Q_R - Q_L\right) \tag{5.12}$$

where $\tilde{A} = \tilde{R}\tilde{\Lambda}\tilde{R}^{-1}$. The matrix R is a matrix constructed from the right eigenvectors of the flux Jacobian, and Λ is a matrix whose diagonal entries contain the absolute values of the eigenvalues of the flux Jacobian. All quantities with the ˜ character denote that they are evaluated at an averaged solution state (between Q_L and Q_R); for this compressible flow system, this state is simply the arithmetic average of Q_L and Q_R.

5.2.2 Viscous Terms

The viscous terms can be discretized by any of several methods; in general, either a finite element or a finite volume technique is employed. Development of the various techniques is given below; for now, it suffices to note that the viscous terms, since they are linear, reduce to simple linear combinations of surrounding vertices (assuming that a nearest-neighbor stencil is maintained):

$$\Re_{0,vis} = \frac{1}{Re} \sum_{i \in \mathcal{N}(0)} C_i \left(Q_i - Q_0 \right) \tag{5.13}$$

where C_i is a matrix containing the coefficients that reflect the viscous behavior. The task of the finite element or finite volume method is to specify these coefficients, which depend on geometry only.

5.2.2.1 Galerkin Finite Element Method

First, the conservation statement is written in differential form:

$$\frac{\partial Q}{\partial t} + \nabla \cdot \vec{F} - \underbrace{\frac{1}{Re} \nabla \cdot \vec{G}}_{viscous\ terms} = 0 \tag{5.14}$$

In this development, only the viscous terms are considered; so, the inviscid term will be neglected. Now, let the linear operator $\mathcal{P}$ be defined as

$$\mathcal{P}(Q) = \frac{\partial Q}{\partial t} - \frac{1}{Re} \nabla \cdot \vec{G} \tag{5.15}$$

The finite element method multiplies the operator equation by a basis function and integrates the result over the spatial domain. In this case, the basis function ϕ is chosen to be a linear function in each element that is defined as unity at the vertex in question, and zero at each of the neighbor vertices. Letting Γ be the union of all elements that intersect vertex 0,

$$\int_{\Gamma} \phi \mathcal{P}(Q) d\Omega = 0 \tag{5.16}$$

Note that the integration is zero outside of the domain of influence of ϕ, because of the definition of the basis function above. The domain of influence of ϕ is simply the union of all elements intersecting vertex 0. Now, the definition for $\mathcal{P}(Q)$ may be inserted, and simultaneously the integration is broken into pieces. Note that this operation is done without approximation:

$$\frac{\partial \bar{Q}_0}{\partial t} \int_\Gamma \phi d\Omega - \frac{1}{Re} \sum_{e \in \mathcal{E}(0)} \int_{\Gamma_e} \phi_e \nabla \cdot \vec{G}_e \, d\Omega = 0 \tag{5.17}$$

Note that $\bar{Q}$ is a volume averaged dependent variable vector, which is equivalent to using a lumped mass matrix (the definition is the same as that proposed in the finite volume method). Now, using the following product rule,

$$\nabla \cdot (\phi \vec{f}) = \phi \nabla \cdot \vec{f} + \vec{f} \cdot \nabla \phi \tag{5.18}$$

a substitution may be made to obtain the following:

$$\frac{\partial \bar{Q}_0}{\partial t} \int_\Gamma \phi d\Omega - \frac{1}{Re} \sum_{e \in \mathcal{E}(0)} \int_{\Gamma_e} \nabla \cdot (\phi_e \vec{G}_e) d\Omega + \frac{1}{Re} \sum_{e \in \mathcal{E}(0)} \int_{\Gamma_e} \vec{G}_e \cdot \nabla \phi_e d\Omega = 0 \tag{5.19}$$

The second term in the above expression can be converted to a surface integral via the divergence theorem. Then, it is trivial to see that these boundary terms are identically zero, since the basis function ϕ is zero on the boundary of the integration (the outer boundary of the union of elements). Using the fact that $\nabla \phi_e$ can be computed exactly if one uses linear elements (a simple application of Green's theorem on a given element proves this),

$$\nabla \phi_e = \frac{-1}{n_{dim} \Gamma_e} \vec{n}_e \tag{5.20}$$

where $\vec{n}_e$ is an outward pointing normal for the exposed face of element ϵ. Equation 5.20 may be immediately substituted into Equation 5.19 to obtain

$$\mathcal{V}_0 \frac{\partial \bar{Q}_0}{\partial t} - \frac{1}{n_{dim} Re} \sum_{e \in \mathcal{E}(0)} \vec{G}_\epsilon \cdot \vec{n}_\epsilon = 0 \tag{5.21}$$

Note that the integral has disappeared, since between Equation 5.19 and Equation 5.21 the constant terms $\vec{G}_e$ and $\vec{n}_e$ are removed from the integral. Also, the integral of the basis function over the domain of integration Γ is equal to $\Gamma / (n_{dim} + 1) = \mathcal{V}$ as long as the control volume is defined using a median dual.

The viscous flux vector $\vec{G}_e$ in turn depends on the solution gradients. So, a means must be provided to evaluate the solution gradient in a particular element, since this is the entity in which $\vec{G}_e$ is evaluated. For a simplical element, Green's theorem may be utilized:

$$\int_{\Gamma_e} \nabla \psi d\Omega = \int_{\partial \Omega_e} \psi \hat{\vec{n}}_e dA \tag{5.22}$$

where ψ is a generic solution variable. Using this expression for evaluating the gradient in an element, it is possible to complete the approximation for the viscous terms.

5.2.2.2 Directional Derivative Method

A primary disadvantage of the traditional finite element method is that geometric data is needed that is nonlocal to a particular edge, unless edge coefficients are stored *a priori* (and the nonlocal maps discarded). However, the storage of these edge coefficients is very expensive, since six coefficients per edge must be stored in a three dimensional discretization. Further, for a finite element method, different integration coefficients are needed depending on the element type in a multielement grid. For general element grids, it is expedient to use only edge-local information to compute the viscous fluxes; this allows the evaluation of the viscous fluxes associated with each face of the control volume without regard to the varying element types of

the mesh. An algorithm in which no element information is used outside of metric computations is termed a "grid transparent" algorithm [26].

To address this difficulty, one can use a finite volume technique with a direct approximation for the gradients at the quadrature points; one such method that uses this approach is termed the "directional derivative" technique [27], [28]. The overall approach is to combine data obtained from the solution gradients at the vertices (which may have already been computed for dependent variable extrapolation) and edge local data to approximate the gradients required in the viscous terms:

$$\nabla Q_{ij} = \nabla Q_{ij,norm} + \nabla Q_{ij,tan} \tag{5.23}$$

Using a directional derivative along the edge to approximate the normal component of the gradient and the average of the nodal gradients (each edge is connected to nodes $i - j$) to approximate the tangential component of the gradient,

$$\left(\nabla Q_{ij} \cdot \hat{s}\right) \hat{s} \approx \frac{Q_j - Q_i}{|\Delta s|} \hat{s} \tag{5.24}$$

$$\left(\nabla Q_{ij} \cdot \hat{t}\right) \hat{t} \approx \overline{\nabla Q} - \left(\overline{\nabla Q} \cdot \hat{s}\right) \hat{s} \tag{5.25}$$

where $\hat{s}$ is a unit vector in the direction of the edge, $\hat{t}$ is a unit vector in a direction normal to the edge, $\overline{\nabla Q} = \frac{1}{2}\left(\nabla Q_i + \nabla Q_j\right)$, and $\vec{\Delta s} = \vec{x}_j - \vec{x}_i$. Combining Equation 5.24 and Equation 5.25 leads to the following formula for the edge gradient:

$$\nabla Q_{ij} \approx \overline{\nabla Q} + \left[Q_j - Q_i - \overline{\nabla Q} \cdot \vec{\Delta s}\right] \frac{\vec{\Delta s}}{|\vec{\Delta s}|^2} \tag{5.26}$$

Since the midpoint of the edge is the quadrature point for the current finite volume method, this approximation may be used directly to evaluate the full viscous flux vector associated with the control volume face. Typically, the weighted least squares method is used to evaluate the nodal gradients in the preceding formula.

Because of the use of vertex gradients, the stencil for this method is no longer a nearest-neighbor stencil. Since the data structures used for storing the sparse matrix cannot typically support stencils larger than nearest-neighbor, part of the residual linearization from the directional derivative method must be neglected on the left hand side. The left hand side terms can potentially include all contributions from nodes i, j, and other nodes in the local stencil, but not contributions from distance-two nodes. In the current approach, all possible contributions in the nearest-neighbor stencil are taken into account.

5.2.3 Higher Order Accuracy

A second order (in space) method for the inviscid terms is constructed by extrapolating the solution at the vertices to the faces of the surrounding control volume. Either Green's theorem or a least squares method is used to compute the gradients at the vertices for the extrapolation. With these gradients known, the variables at the interface are computed as

$$Q_{ij} = Q_0 + \nabla Q_0 \cdot \vec{r} \tag{5.27}$$

where $\vec{r}$ is defined as $\vec{x}_f - \vec{x}_0$; the position $\vec{x}_f$ is the midpoint of the edge (the quadrature point for the control volume face). To compute the gradient via Green's theorem, the following simple formula is used:

$$\int_{\Omega_0} \nabla Q d\mathcal{V} = \int_{\partial\Omega_0} Q\hat{\vec{n}} dA \tag{5.28}$$

Assuming a linear distribution in each element and noting that the area of integration is made up of discrete pieces,

$$\nabla Q_0 = \frac{1}{\mathcal{V}_0} \sum_{i \in \mathcal{N}(0)} \frac{1}{2}(Q_0 + Q_i) \tag{5.29}$$

This formula is equally applicable in two and three dimensions; of course, special consideration must be taken at the boundaries such that a constant gradient is recovered if a linear distribution is input.

5.3 Temporal Discretization

After the spatial terms have been suitably discretized, the time derivative term appearing in Equation 5.10 must be approximated. A general difference expression is available for this purpose ([29], [30]), and is given as follows:

$$\Delta q^n = \frac{\theta_1 \Delta t}{1+\theta_2}\frac{\partial}{\partial t}(\Delta q^n) + \frac{\Delta t}{1+\theta_2}\frac{\partial}{\partial t}(q^n) + \frac{\theta_2}{1+\theta_2}\Delta q^{n-1} \tag{5.30}$$

where $\Delta q^n = q^{n+1} - q^n$. A first order accurate in time Euler implicit scheme is given by the choices $\theta_1 = 1, \theta_2 = 0$. Correspondingly, a second order time accurate Euler implicit scheme is given by $\theta_1 = 1, \theta_2 = 1/2$. Since $\theta_1 = 1$ for both time discretizations used in this work, Equation 5.30 can be further simplified:

$$\Delta q^n = \frac{\Delta t}{1+\theta_2}\frac{\partial}{\partial t}\left(q^{n+1}\right) + \frac{\theta_2}{1+\theta_2}\Delta q^{n-1} \tag{5.31}$$

Using Equation 5.10 to replace the time derivative,

$$\frac{\Delta q^n - \frac{\theta_2}{1+\theta_2}\Delta q^{n-1}}{\Delta t} = -\frac{1}{1+\theta_2}\Re^{n+1} \tag{5.32}$$

By the definition of q, one can write $q = \bar{Q}\mathcal{V}$. Then, the following two identities can be formed:

$$\Delta q^n = (\bar{Q}\mathcal{V})^{n+1} - (\bar{Q}\mathcal{V})^n = \mathcal{V}^{n+1}\Delta\bar{Q}^n + \bar{Q}^n\Delta\mathcal{V}^n \tag{5.33}$$

$$\Delta q^{n-1} = (\bar{Q}\mathcal{V})^n - (\bar{Q}\mathcal{V})^{n-1} = \mathcal{V}^{n-1}\Delta\bar{Q}^{n-1} + \bar{Q}^n\Delta\mathcal{V}^{n-1} \tag{5.34}$$

Inserting the above two identities into Equation 5.32, one arrives at the following expression:

$$\frac{\mathcal{V}^{n+1}\Delta\bar{Q}^n - \frac{\theta_2}{1+\theta_2}\mathcal{V}^{n-1}\Delta\bar{Q}^{n-1}}{\Delta t} + \bar{Q}^n\left[\frac{\Delta\mathcal{V}^n - \frac{\theta_2}{1+\theta_2}\Delta\mathcal{V}^{n-1}}{\Delta t}\right] + \frac{1}{1+\theta_2}\Re^{n+1} = 0 \tag{5.35}$$

Now, one must consider the Geometric Conservation Law (GCL). This statement relates the rate of change of a physical volume to the motion of the volume faces:

$$\frac{\partial \mathcal{V}}{\partial t} = \int_{\Omega} \nabla \cdot \vec{V}_s d\mathcal{V} = \int_{\partial\Omega} \vec{V}_s \cdot \hat{\vec{n}} dA \tag{5.36}$$

According to Thomas and Lombard [31] and later Janus [32], the solution of the volume conservation equation must be performed in exactly the same manner as the flow equations to ensure that GCL is satisfied. This procedure ensures that spurious source terms caused by volume changes are eliminated. Using the same time differencing expression (Equation 5.31) to approximate Equation 5.36,

$$\frac{\Delta \mathcal{V}^n - \frac{\theta_2}{1+\theta_2}\Delta \mathcal{V}^{n-1}}{\Delta t} = \frac{1}{1+\theta_2} \Re_{\text{GCL}}^{n+1} \tag{5.37}$$

where $\Re_{\text{GCL}}^{n+1} = \sum_{i \in \mathcal{N}(0)} \vec{\mathcal{V}}_{0i}^{n+1} \cdot \vec{n}_{0i}^{n+1}$. Note that the left hand side of the preceding equation is exactly the bracketed term in Equation 5.35. Replacing the bracketed term and rearranging slightly gives the final form of the discretization of the time derivative:

$$\frac{(1+\theta_2)\mathcal{V}^{n+1}\Delta \bar{Q}^n - \theta_2 \mathcal{V}^{n-1}\Delta \bar{Q}^{n-1}}{\Delta t} + \bar{Q}^n \Re_{\text{GCL}}^{n+1} + \Re^{n+1} = 0 \tag{5.38}$$

5.4 Time Evolution

The final version of the time discretization of the governing equations (Equation 5.38) indicates that the spatial residual must be evaluated at time level $n+1$. Obviously, the solution state at this time level is unknown; to solve this nonlinear equation, one must linearize about the known solution Q^n. One technique for doing so is to use Newton's method; following Equation 5.38, let

$$\Im_0^{n+1}\left(\bar{Q}^{n+1}\right) = \frac{(1+\theta_2)\mathcal{V}_0^{n+1}\Delta \bar{Q}_0^n - \theta_2 \mathcal{V}_0^{n-1}\Delta \bar{Q}_0^{n-1}}{\Delta t} + \bar{Q}^n \Re_{0,\text{GCL}}^{n+1} + \Re_0^{n+1} \tag{5.39}$$

The quantity $\Im^{n+1}$ is the function that should be driven to zero by the Newton iteration. Expanding $\Im^{n+1}$ in a Taylor series from a known level $n+1, m$,

$$\Im_0^{n+1,m+1} = \Im_0^{n+1,m} + \frac{\partial \Im_0}{\partial t}^{n+1,m} \Delta t + \mathcal{O}(\Delta t^2) \tag{5.40}$$

Dropping the $\mathcal{O}(\Delta t^2)$ error term, utilizing the chain rule, and replacing $\frac{\partial \bar{Q}}{\partial t}$ with a first-order difference,

$$\Im_0^{n+1,m+1} \approx \Im_0^{n+1,m} + \frac{\partial \Im_0}{\partial \bar{Q}}^{n+1,m} \Delta \bar{Q}^{n+1,m} \tag{5.41}$$

Since the LHS of the above equation is zero at Newton convergence,

$$-\Im_0^{n+1,m} = \frac{\partial \Im_0}{\partial \bar{Q}}^{n+1,m} \Delta \bar{Q}^{n+1,m} \tag{5.42}$$

where $\Delta \bar{Q}^{n+1,m} = \bar{Q}^{n+1,m+1} - \bar{Q}^{n+1,m}$. Now, expanding the terms and performing the required differentiations of $\Im$ results in the following expression for Newton's method:

$$-\left[\frac{(1+\theta_2)\mathcal{V}_0^{n+1}(\bar{Q}_0^{n+1,m} - \bar{Q}_0^n) - \theta_2 \mathcal{V}_0^{n-1}\Delta \bar{Q}_0^{n-1}}{\Delta t} + \bar{Q}_0^n \Re_{0,\mathrm{GCL}}^{n+1} + \sum_{i \in \mathcal{N}(0)} \vec{H}_{0i}^{n+1,m} \cdot \vec{n}_{0i}^{n+1}\right] =$$

$$\left[\frac{(1+\theta_2)\mathcal{V}_0^{n+1} I}{\Delta t} + \sum_{i \in \mathcal{N}(0)} \frac{\partial \vec{H}_{0i}^{n+1,m} \cdot \vec{n}_{0i}^{n+1}}{\partial \bar{Q}_0}\right] \Delta \bar{Q}_0^{n+1,m} +$$

$$\sum_{i \in \mathcal{N}(0)} \left[\frac{\partial \vec{H}_{0i}^{n+1,m} \cdot \vec{n}_{0i}^{n+1}}{\partial \bar{Q}_i} \Delta \bar{Q}_i^{n+1,m}\right] \tag{5.43}$$

where $\Delta \bar{Q}_0^{n-1} - \bar{Q}_0^n - Q_0^{n-1}$. For notational convenience, both the inviscid and viscous terms are collapsed into a single flux function H. Note that the iteration can be started by using an initial guess of $\bar{Q}^{n+1,0} = \bar{Q}^n$. Also, performing only one iteration of Newton's method per time step (with 1st order time discretization and no GCL terms) is equivalent to a time linearization of the spatial terms only. However, writing the method in this framework is more general than a straightforward time linearization of the nonlinear terms.

It is clear that Equation 5.43 gives rise to an algebraic sparse matrix system. If the matrix system is written as $\mathcal{A}x = b$, the left side of the above equation represents b, $\Delta\bar{Q}$ represents x, and the coefficients preceding $\Delta\bar{Q}$ represent the sparse matrix $\mathcal{A}$. Furthermore, the coefficients leading $\Delta\bar{Q}_0$ make up the diagonal of $\mathcal{A}$, and the coefficients leading $\Delta\bar{Q}_i$ are the off-diagonal elements of $\mathcal{A}$.

Various methods are available to solve this sparse system of equations; direct methods, however, are impractical due to an operation count of $\mathcal{O}(Nb_w^2)$, where b_w is the half-bandwidth of the matrix. Iterative methods hold more promise in terms of practicality, and can be loosely divided into matrix splitting relaxation methods and gradient-based techniques. In the present solution algorithm, the Jacobi method (relaxation) and symmetric Gauss-Seidel method (relaxation) as techniques to solve the linear system are investigated.

5.4.1 Jacobi Iteration

The Jacobi iterative solver splits the matrix into a diagonal, upper triangular, and lower triangular part:

$$\mathcal{A} = [\mathcal{L} + \mathcal{D} + \mathcal{U}] \tag{5.44}$$

and defines the iteration as follows:

$$\mathcal{D}\Delta\bar{Q}^{n+1,m+1,k+1} = \Re^{n+1,m} - [\mathcal{L} + \mathcal{U}]\,\Delta\bar{Q}^{n+1,m+1,k} \tag{5.45}$$

where $\Delta\bar{Q}^{n+1,m+1,k+1} = Q^{n+1,m+1,k+1} - Q^{n+1,m}$. The advantage of this solution method is that the matrix multiply $[\mathcal{L} + \mathcal{U}]\Delta\bar{Q}^{n+1,m+1,k}$ is very easy to carry out; however, the primary disadvantage is that the method typically yields very slow convergence. Implementation of this technique uses a single loop over the edge structures to multiply the off-diagonal terms, followed by a backsubstitution to solve $\Delta\bar{Q}^{n+1,m+1,k+1} = \mathcal{D}^{-1}RHS$. A variant of this algorithm is to use "coloring" such that the convergence of the algorithm is improved. For example, if one colors all odd-numbered nodes "red" and all even-numbered nodes "black", the following Red-Black

Jacobi scheme is given (sometimes given the misnomer of a Red-Black Gauss-Seidel scheme):

$$[\mathcal{D}]\,\Delta\bar{Q}_R^{n+1,m+1,k+1} = \Re^{n+1,m} - [\mathcal{L}+\mathcal{U}]\left(\Delta\bar{Q}_R^{n+1,m+1,k} \cup \Delta\bar{Q}_B^{n+1,m+1,k}\right) \tag{5.46}$$

$$[\mathcal{D}]\,\Delta\bar{Q}_B^{n+1,m+1,k+1} = \Re^{n+1,m} - [\mathcal{L}+\mathcal{U}]\,(\Delta\bar{Q}_R^{n+1,m+1,k+1} \cup \Delta\bar{Q}_B^{n+1,m+1,k}) \tag{5.47}$$

where the subscripts R and B denote the portions of the reference vector which belong to the red and black sets, respectively. As the number of colors approaches the number of nodes, the colored Jacobi scheme approaches the unidirectional Gauss-Seidel scheme.

5.4.2 Symmetric Gauss-Seidel Iteration

The symmetric Gauss-Seidel (SGS) matrix solution method begins by splitting the matrix into an upper and lower triangular part:

$$\mathcal{A} = [\mathcal{L}+\mathcal{D}+\mathcal{U}] \tag{5.48}$$

Where the diagonal, upper, and lower operators are defined by

$$\mathcal{D} = \left[\frac{(1+\theta_2)\mathcal{V}_0 I}{\Delta t} + \sum_{i\in\mathcal{N}(0)} \frac{\partial \vec{H}_{0i}^{n+1,m}\cdot\vec{n}_{0i}^{n+1}}{\partial\bar{Q}_0}\right](\cdot) \tag{5.49}$$

$$\mathcal{U} = \sum_{i\in\mathcal{N}_U(0)} \frac{\partial \vec{H}_{0i}^{n+1,m}\cdot\vec{n}_{0i}^{n+1}}{\partial\bar{Q}_i}(\cdot) \tag{5.50}$$

$$\mathcal{L} = \sum_{i\in\mathcal{N}_L(0)} \frac{\partial \vec{H}_{0i}^{n+1,m}\cdot\vec{n}_{0i}^{n+1}}{\partial\bar{Q}_i}(\cdot) \tag{5.51}$$

Then, the symmetric Gauss-Seidel method can be written as the following two-step process per iteration:

$$[\mathcal{L}+\mathcal{D}]\,\Delta\bar{Q}^{n+1,m+1,k+\frac{1}{2}} + [\mathcal{U}]\,\Delta\bar{Q}^{n+1,m+1,k} = \Re^{n+1,m} \tag{5.52}$$

$$[\mathcal{D}+\mathcal{U}]\,\Delta\bar{Q}^{n+1,m+1,k+1} + [\mathcal{L}]\,\Delta\bar{Q}^{n+1,m+1,k+\frac{1}{2}} = \Re^{n+1,m} \tag{5.53}$$

Typically, the initial guess for $\Delta\bar{Q}^{n+1,m+1,0}$ is zero. The implementation of this algorithm is particularly simple; for the first pass, sweep forward through the vertices. For every vertex, multiply the off-diagonal terms by the most recent solution stored in a buffer $\Delta\bar{Q}$, subtract them from the corresponding element in $\Re^{n+1,m}$, solve the system (5x5 system, for 3D compressible flows) $\mathcal{D}x = R\bar{H}\bar{S}$, and copy the solution back into the $\Delta\bar{Q}$ buffer. For the second sweep, perform exactly the same operations but instead loop backward though the vertices instead of forward. Note that the Gauss-Seidel algorithm requires a vertex to surrounding edge map, since the sparse matrix-vector multiplies must be undertaken on a row-by-row basis.

5.5 Turbulence Modeling

A model for the effects of turbulence is a necessary component for simulating high Reynolds number flows. In the present work, the turbulence model is incorporated in a "loosely-coupled" procedure; that is, the mean flow equations are solved first, and then the turbulence model is solved independently. Coupling between the two is accomplished since the turbulence model uses the most recently computed solution (Q^{n+1}), and the solution of the core governing equations uses the most recently computed eddy viscosity (μ_t^n). Figure 5.2 outlines this procedure.

5.5.1 Spalart-Allmaras Turbulence Model

The one-equation turbulence model of Spalart and Allmaras is available for high Reynolds number flows [33]; this model formulates a transport equation for the turbulent Reynolds number, which is then related to the turbulent viscosity. From the original Spalart and Allmaras paper [33], a transport equation can be written for a working variable $\tilde{\nu}$ (the turbulent Reynolds number):

$$\underbrace{\frac{\partial \tilde{\nu}}{\partial t}}_{\text{change}} + \underbrace{\vec{V}\cdot\nabla\tilde{\nu}}_{\text{convection}} = \underbrace{c_{b1}\left[f_{r1} - f_{t2}\right]\tilde{S}\tilde{\nu}}_{\text{production}} + \underbrace{\frac{1}{\sigma}\left[\nabla\cdot\left((\nu+\tilde{\nu})\nabla\tilde{\nu}\right) + c_{b2}\left(\nabla\tilde{\nu}\right)^2\right]}_{\text{diffusion}} -$$

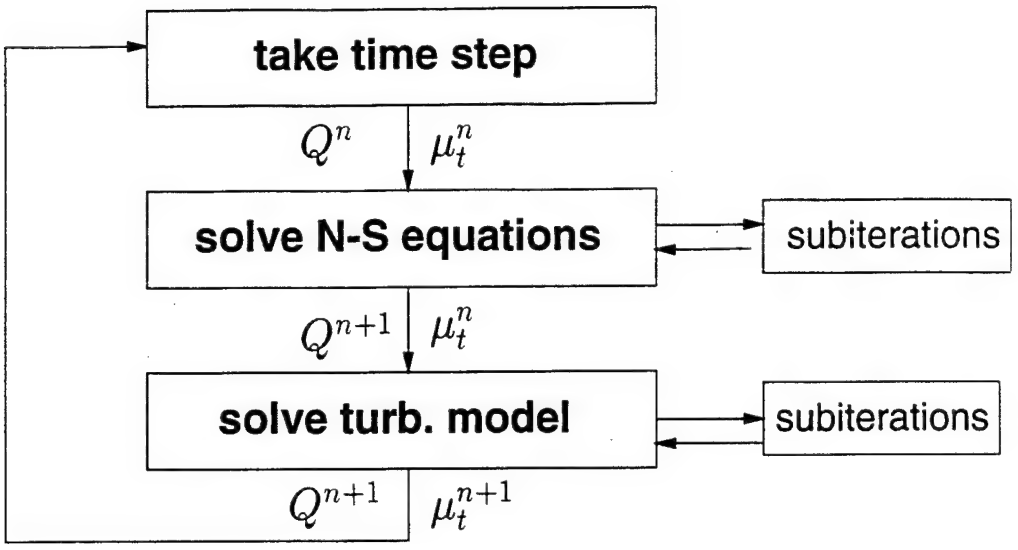


Figure 5.2: The solution procedure that incorporates the turbulence model is decoupled from the original system of equations.

$$\underbrace{\left[c_{w1} f_w - \frac{c_{b1}}{\kappa^2} f_{t2}\right]\left[\frac{\tilde{\nu}}{d}\right]^2}_{\text{destruction}} + \underbrace{f_{t1}\Delta U^2}_{\text{trip}} \tag{5.54}$$

For evaluation of the diffusive term, it is useful to slightly rearrange Equation 5.54 into an equivalent form. Also, in this work, the trip term is neglected. After nondimensionalization, the equation for $\tilde{\nu}$ becomes

$$\frac{\partial \tilde{\nu}}{\partial t} + \vec{V}\cdot\nabla\tilde{\nu} = c_{b1}\left[f_{r1} - f_{t2}\right]\tilde{S}\tilde{\nu} - \frac{1}{Re}\left[c_{w1} f_w - \frac{c_{b1}}{\kappa^2} f_{t2}\right]\left[\frac{\tilde{\nu}}{d}\right]^2 + \frac{1}{\sigma Re}\left\{\nabla\cdot\left[\left(\nu + (1+c_{b2})\,\tilde{\nu}\right)\nabla\tilde{\nu}\right] - c_{b2}\tilde{\nu}\nabla\cdot\left[\nabla\tilde{\nu}\right]\right\} \tag{5.55}$$

where the function definitions in nondimensionalized form are

$$\nu_t = \tilde{\nu} f_{v1} \tag{5.56}$$

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3} \tag{5.57}$$

$$\chi = \frac{\tilde{\nu}}{\nu} \tag{5.58}$$

$$\tilde{S} = S + \frac{\tilde{\nu}}{Re\kappa^2 d^2} f_{v2} \tag{5.59}$$

$$S = |\vec{\omega}| \tag{5.60}$$

$$f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}} \tag{5.61}$$

$$f_w = g\left[\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right]^{1/6} \tag{5.62}$$

$$g = r + c_{w2}(r^6 - r) \tag{5.63}$$

$$r = \frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2 Re} \tag{5.64}$$

$$f_{t2} = c_{t3} \exp\left(-c_{t4}\chi^2\right) \tag{5.65}$$

$$f_{r1} = \begin{cases} 1 & : \text{ default} \\ (1 + c_{r1}) \frac{2r^*}{1 + r^*} \left[1 - c_{r3} \tan^{-1}\left(c_{r2}\tilde{r}\right)\right] - c_{r1} & : \text{ modified [34]} \end{cases} \tag{5.66}$$

$$r^* = \sqrt{\frac{1}{2}\mathscr{S}}/|S| \qquad \text{where } \mathscr{S} = [u_{i,j} + u_{j,i}][u_{i,j} + u_{j,i}] \tag{5.67}$$

$$\tilde{r} = (1 - r^*)/2 \tag{5.68}$$

The constant definitions are as follows:

$$\kappa = 0.41 \qquad \sigma = 2/3 \qquad c_{v1} = 7.1$$

$$c_{b1} = 0.1355, \qquad c_{b2} = 0.622,$$

$$c_{w1} = \frac{c_{b1}}{\kappa^2} + (1 + c_{b2})/\kappa \qquad c_{w2} = 0.3 \qquad c_{w3} = 2.0,$$

$$c_{t1} = 1.0 \qquad c_{t2} = 2.0 \qquad c_{t3} = 1.1 \qquad c_{t4} = 2.0$$

$$c_{r1} = 1 \qquad c_{r2} = 12 \qquad c_{r3} = 1$$

Note that Equation 5.66 gives two options for computing f_{r1}. The first option is the default value from the original formulation of the model [33]. The second option is a modification to the production term suggested in [34] to better preserve vortices in the near field. Unless otherwise stated, simulations are performed in the present work using the default option.

Equation 5.55 must be appropriately discretized for implementation into the unstructured solution procedure. A Galerkin finite-element method or a directional derivative method (Section 5.2.2) is used to discretize the diffusive terms that are inside of the divergence operator; the $\tilde{\nu}$ term outside of the divergence is assumed to be constant within the control volume. A pure upwind method is used to discretize the convective terms. The turbulence production and destruction terms are evaluated with the assumption that $\tilde{\nu}$ is constant within the given control volume. Each of the terms are appropriately linearized with respect to time (and attention paid to positivity considerations) to derive the sparse matrix required for implicit solution of the governing equation.

On no-slip surfaces, the turbulent Reynolds number ($\tilde{\nu}$) is defined to be zero and therefore is not solved for. At farfield inflow boundaries, $\tilde{\nu}$ is set to a freestream value of 1/10 (as per the recommendation of [33]) for the boundary face flux evaluation. The dependent variable is extrapolated from the interior for the corresponding flux evaluation on farfield outflow boundaries.

5.6 Parallel Methodology

The present investigation explores several alternative methods of treating the domain decomposition, subdomain interface connectivity, and subdomain coupling in the parallel unstructured solution algorithm. In addition, proper embedding of the parallelization within the iteration hierarchy is discussed. The discussions concerning the iteration hierarchy, subdomain coupling, and interface connectivity apply primarily to the mean flow, but are also equally applicable to the solution of the turbulence model.

5.6.1 Iteration Hierarchy

The notion of an iteration hierarchy was introduced in [35]. The purpose of the levels of the iteration hierarchy is to reduce error components that arise at various stages of the solution process. The Newton iteration reduces errors arising from the time linearization of the nonlinear terms, while the inner subiterations reduce error that is caused by the splitting of the linear system. In addition, these linear subiterations coupled with subdomain communication can be used to eliminate errors that occur due to the loss of coupling caused by the partitioning of the domain into subdomains.

The sequential iteration hierarchy utilized in [36] greatly reduces memory requirements by sequential solution within each subdomain. The procedure allows reuse of storage for Jacobians and other memory intensive operations, and allows for one update of all communicated quantities at the beginning of each time step.

Since parallel concurrency allows one to update at any desired point in the iteration hierarchy, it is possible to relax the restrictions of the hierarchy given in [36] to formulate a more flexible updating strategy. Reorganization of the updating structure is intended to address shortcomings of the sequential iteration hierarchy:

- since gradients are updated between time steps, gradients on the interface are effectively lagged. Thus, the residual is computed using n time level gradients on the interior, but $n-1$ time level gradients on the block interfaces. This leads to an inconsistency in the residual computations at the subdomain interfaces.
- since the solution Q is updated at the beginning of the time step, it is impossible to carry out Newton iterations, if desired.
- no ΔQ information is passed during sparse matrix solves, which forces a non-implicit handling of the subdomain interfaces. As the number of subdomains increase, the sparse matrix solver suffers from convergence degradation.

The proposed iteration hierarchy for the present work is shown in Figure 5.3, where two updating modes are defined: concurrent subdomain iteration and sequential subdomain iteration. The sequential mode of subdomain iteration is the same as that presented in [36]. In concurrent mode, updating is carried out such that interface quantities are updated at the same time as they

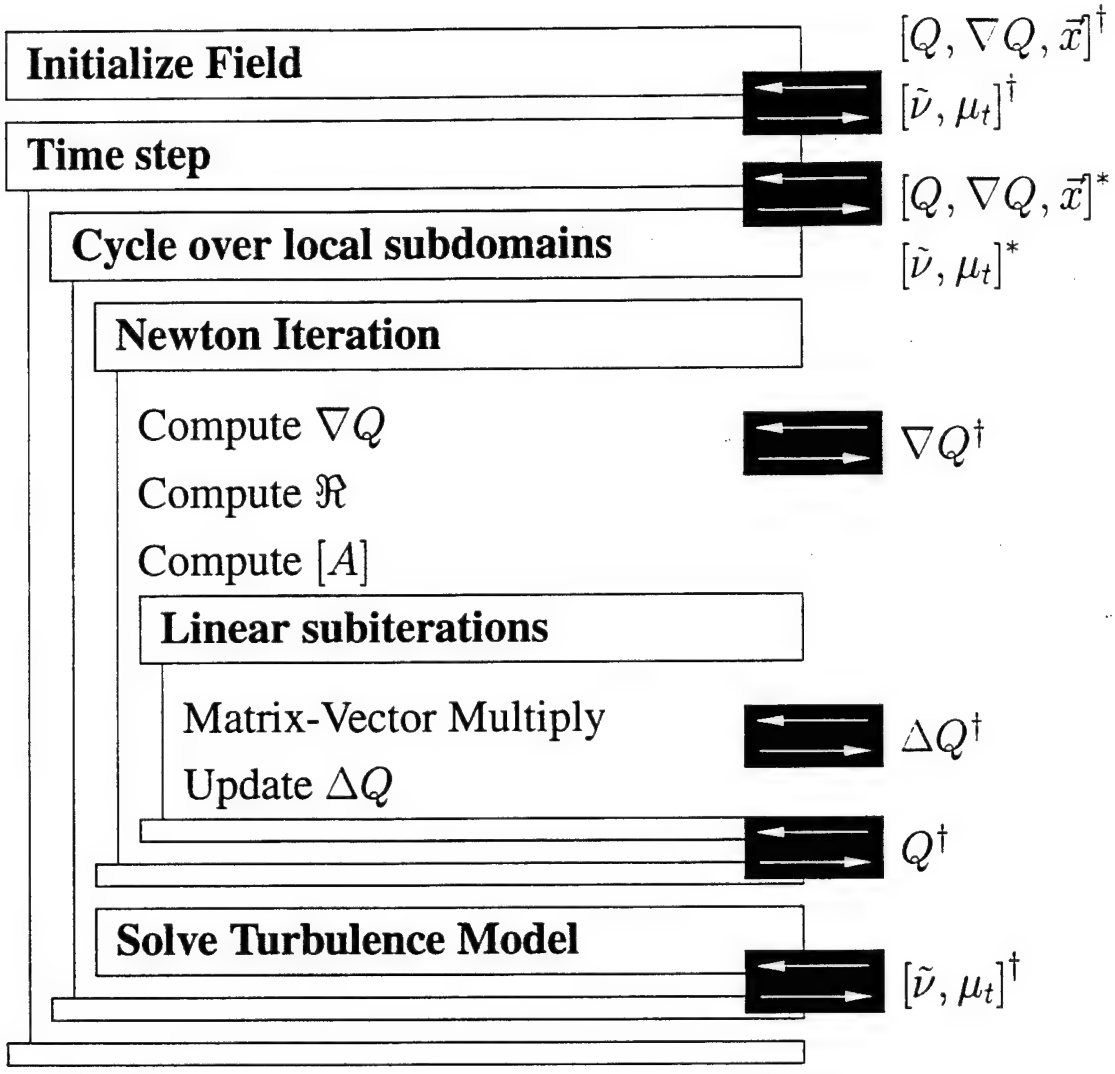
would be in a single subdomain algorithm. This provides the correct time and/or iterative level for each of the quantities that is exchanged via communication.

Two items should be addressed by the iteration hierarchy in a parallel context: 1) consistency of the residual and 2) alleviation of degradation in convergence.

Residual consistency ensures that the residual computed for every node in the domain is the same regardless of the number of subdomains. Since the residual is a function of $\vec{x}$, Q, and ∇Q, it is sufficient to make copies of these data available on the subdomain interfaces such that the computed residual on each side of these interfaces is the same. Alternatively, one subdomain can be responsible for computing the residual for an interface node, and this residual is then communicated to the other. The concurrent subdomain iteration relies on the former technique, which is to distribute each quantity that is used in the residual calculation ($\vec{x}$, Q, ∇Q) and compute the interface fluxes redundantly in each subdomain. Because the present solution algorithm uses coarse-grained parallelism (the computational volume is large compared to the communication surface), the cost of this redundancy is of small consequence.

Secondly, the hierarchy should allow for the communication necessary for the parallel algorithm to display convergence characteristics reasonably close to its serial counterpart. In concurrent subdomain iteration, this is accomplished by allowing communication of ΔQ during the subiterations so that the parallel algorithm is able to access the entire vector corresponding to the complete domain rather than just the vector belonging to a particular subdomain. In this way, sparse matrix-vector multiplies can be carried out in a parallel context that produce the same results as in a sequential context.

Note that the sequential updating mode given in Figure 5.3 does not allow for residual consistency or alleviation of convergence degradation. Although this characteristic is not desirable, this variation of the hierarchy is present to support the memory leveraging procedure published in [36]. Using this variation, solutions may be performed using (typically) 1/5 of the memory consumed by the serial algorithm, since one is able to cycle each block in sequence and reuse memory abandoned by the last subdomain in the cycle. The sequential updating mode



* sequential updating hierarchy
† concurrent updating hierarchy

Figure 5.3: Sequential and concurrent iteration hierarchies

also allows one to map more than one subdomain onto a particular processor. This capability is available at the cost of some convergence degradation and can be used for steady flows only, where the residual inconsistency on the interfaces is inconsequential at algorithm convergence.

5.6.2 Subdomain Interface Treatments

Two types of subdomain interfaces are considered. The first is a surface interface in which the decomposition is along a surface made up of element faces, as shown in Figure 5.4; this is the connectivity technique employed by Sheng and Whitfield [36]. The second is a mesh vertex decomposition in which distinct node-based control volumes are assigned to subdomains as in Figure 5.5; the interface thus lies on the dual to the mesh. Each technique entails a different storage and communication paradigm.

A clarification should be issued here in regard to the terminology used in this work; whereas "element-based" and "vertex-based" decompositions indicate directly the entity that is separated between the subdomains, a "control-volume-based" decomposition is slightly ambiguous. In a node-based solution scheme (as in the current work, control volumes are constructed around each node), a control volume decomposition is equivalent to a nodal decomposition, since a vertex and a control volume have a one-to-one correspondence. However, in a cell-based solution scheme, a control volume decomposition corresponds to an element-based connectivity scheme, since each element corresponds to one control volume (as in [37]). The two types of decompositions discussed here, the element-based and node-based decomposition, are presented in the context of a nodal control volume solution technique.

5.6.3 Element-based Connectivity Scheme

To establish an element-based subdomain connectivity, "phantom" entities are created for every primitive in the mesh. This involves creating lists of phantom nodes, elements, boundary facets, and edges. It is important to note that since elements are assigned uniquely to each partition, nodes on a subdomain interface are duplicated in at least two blocks; thus, the ownership status of these nodes is ambiguous. A diagram of the element-based connectivity

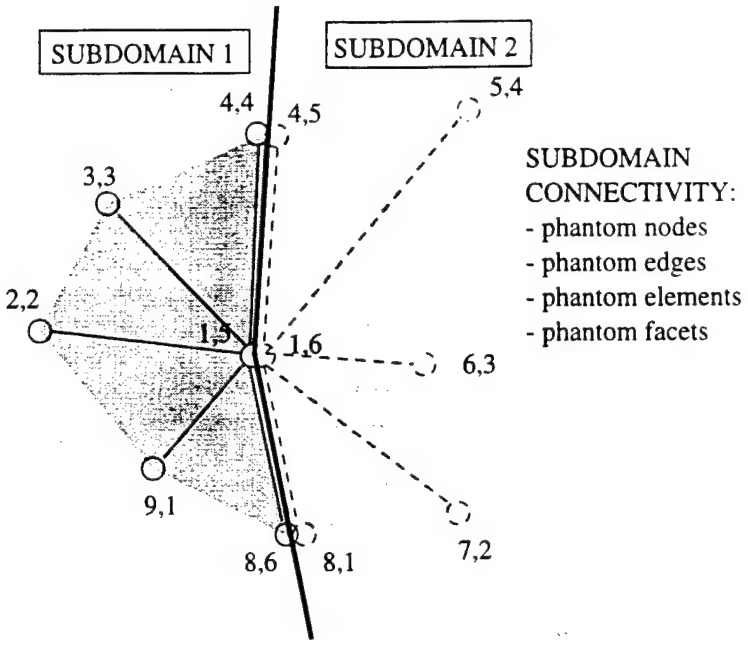


Figure 5.4: Schematic of the element-based connectivity scheme

scheme is shown in Figure 5.4. Note that global node numberings are listed first, and subdomain local numberings are listed second.

In a given subdomain, phantom nodes are created for every vertex on the subdomain interface as well as for every node connected to the interface in the bordering subdomain. To maintain connectivity for these newly created vertices, it is necessary to construct phantom elements, phantom boundary facets, and phantom edges. Using these newly created entities, one can treat the connectivity structure as a layer of interior entities that can distribute or accumulate computed quantities as needed. Note that this treatment leads to a distance-two overlap of the subdomains.

To compute residual and Jacobian contributions from adjacent subdomains, loops are constructed over phantom entities which perform exactly the same operations as the corresponding loops over the physical mesh entities. Therefore, each loop in the solver is followed by a nearly identical loop over the associated phantom primitive. In these secondary loops, tests are performed on the nodes owned by the phantom entities such that data scatters are undertaken appropriately. A pseudocode example (for computing the inviscid residual) is below:

```
do i = 1,nedge
  n1 = node 1 of edge i
  n2 = node 2 of edge i
  gather geometry and solution information for i, n1, n2
  compute flux along edge i
  add flux to residual for n1
  subtract flux from residual for n2
enddo

do i = 1,neph
  nph1 = phantom node of phantom edge i
  nph2 = phantom node of phantom edge i
  n1 = node associated with nph1
  n2 = node associated with nph2
  gather geometry and solution information for i, nph1, nph2
  compute flux along phantom edge i
  if n1 exists, add flux to residual for n1
  if n2 exists, subtract flux from residual for n2
enddo
```

5.6.4 Node-based Connectivity Scheme

An alternative method to handle the subdomain interfaces is to uniquely assign each vertex to a particular subdomain; this treatment is termed a node-based interface scheme. A diagram of this interface connectivity scheme is shown in Figure 5.5. Note that global node numberings are listed first, and subdomain local node numberings are listed second. Using this treatment, only phantom nodes must be created to fully define the connectivity between each subdomain.

Using this control-volume based connectivity scheme, each subdomain appears to the solver as a complete domain, except that phantom nodes may exist. Normal entities in the grid, such as elements and edges, may contain one or more phantom nodes. The only special treatment given to these nodes is that accumulations (if they are performed) are ignored, and these vertices are the points at which incoming parallel communications take place. Outgoing communications take place from any nodes connected to a phantom node. Defining the subdomain connectivity in this way leads to a distance-one overlap between subdomains. To compute residual and Jacobian contributions across subdomains, no modification to the core computation is required (in contrast the the element-based connectivity scheme).

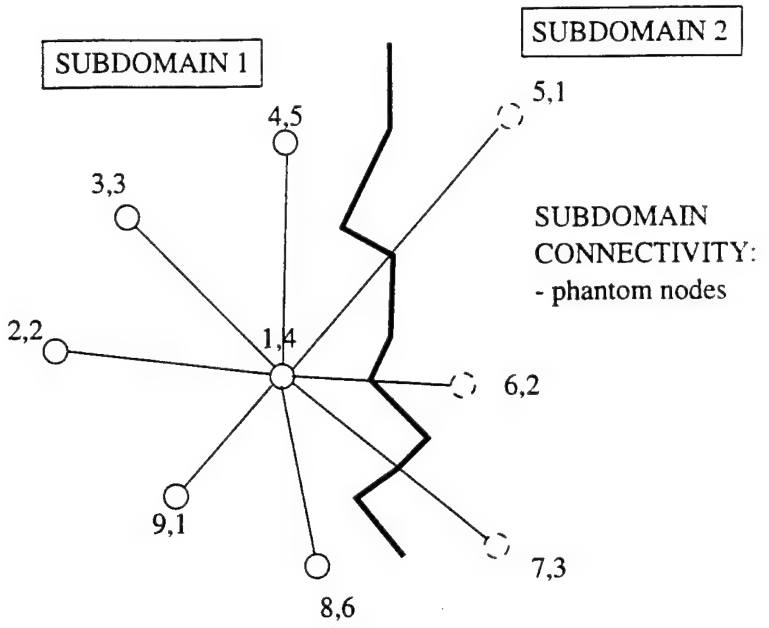


Figure 5.5: Schematic of the node-based connectivity scheme

To compute residual and Jacobian contributions across subdomains, only slight modification to the original code has been performed in this work. The modification consists of inserting a check before any accumulation to a node takes place, to ensure that the node accumulated to is not a phantom node. This check is not strictly necessary (since accumulations to a phantom node will be ignored), but is implemented for clarity as well as efficiency purposes. The implementation is demonstrated by the following pseudocode:

```
do i = 1,nedge
  n1 = node 1 of edge i
  n2 = node 2 of edge i
  gather geometry and solution information for i, n1, n2
  compute flux along edge i
  if n1 is not phantom, add flux to residual for n1
  if n2 is not phantom, subtract flux from residual for n2
enddo
```

Algorithmic, memory, and execution time issues are associated with the two connectivity scheme types, and it has been found that the node-based connectivity scheme has several advantages:

- improved convergence properties
- problem size does not (artificially) increase as number of partitions are increased; hence, the algorithm can now be scalable
- less computation per time step
- eliminates solution ambiguity, since a specific subdomain owns each control volume and hence its associated volume averaged solution variables
- amount of connectivity information decreases; thus, memory is conserved
- handles general element grids and other definitions of nodal control volumes without modification

For edge-based computations (primarily the residual and Jacobian) special consideration must be given to the edges that span interpartition boundaries (for ease of exposition, the residual calculations are considered here). The difficulty arises since flux integrals on nodes adjacent to partition boundaries cannot be completed without some sort of communication. For these edges, flux computations may be 1) calculated in a preassigned subdomain and subsequently communicated to the neighboring subdomain, or 2) calculated redundantly in each subdomain. Thus, the optimal choice depends on whether the speed of communicating the interface fluxes is greater than the speed of computing them. In this work, the second option is chosen due to 1) in most situations, computing a flux for an edge consumes less overall time than communicating the result of a flux evaluation, and 2) message passing for edges would entail the building and maintenance of a secondary (expensive) subdomain connectivity structure. It should be noted that even if the flux values are communicated instead of redundantly computed, a preliminary communication to/from phantom nodes is still necessary to provide the state vectors for the Riemann problem. In a field solver where the residual (and Jacobian) calculations are extremely costly to compute (such as in chemically reacting flows), the most efficient choice could be to communicate flux evaluations on interpartition edges rather than redundantly compute them.

5.6.5 Subdomain Iteration Methods

A primary advantage of implicit methods is the global communication of data that occurs each time step. However, the division of a domain into subdomains implies that, without

specific procedures to ensure subdomain coupling, this global communication degenerates into propagation of waves only within the individual subdomains. Obviously, the loss of global wave propagation leads to a deterioration in convergence characteristics of the parallel algorithm when compared to the original serial algorithm.

One can loosely view the solution of the sparse linear system (arising from an implicit approximation) as a sequence of sparse-matrix vector products. Each task owns a set of rows of the sparse matrix and the corresponding section of a vector to be multiplied. However, a given task does not necessarily own the entire section of the vector that must be multiplied by a given row of the sparse matrix; so, one must provide a mechanism to pass these nonlocal entries of the vector from the owning task to the task that must use the quantity in the multiply operation. Storage locations are set aside for this procedure as described in Section 5.6.2.

The concurrent iteration hierarchy given in Section 5.6.1 allows for proper updating to take place during the sparse-matrix vector multiplies (linear subiterations). It is this communication which provides the subdomain coupling necessary to approximately maintain the convergence rate of the serial implicit algorithm. If the contributions from an adjacent subdomain's control volumes are neglected (termed a block Jacobi [BJ] subdomain coupling method), this places an artificial boundary within the domain from which no useful information propagates and no useful information can penetrate. Hence, convergence is degraded by the presence of subdomain boundaries in the domain unless special treatment of these interfaces is undertaken. Given that a relaxation method is used on the interior, the BJ technique degenerates into a traditional point Jacobi method as the number of subdomains approaches the number of nodes in the domain.

For relaxation algorithms (such as Gauss-Seidel), this updating during subiterations now allows recovery of a modified form of the original algorithm. The degree to which the relaxation algorithm is recovered is determined by the frequency of updating of the interface ΔQ's. A weak coupling can be accomplished by updating ΔQ only after each subiteration, and maximum coupling can be accomplished by updating ΔQ at every available point; in this work, one has the

opportunity to update between color changes (BGS3), directional sweeps (BGS2, BGS3), and after each subiteration (BGS1, BGS2, BGS3).

To establish terminology for the different levels of subdomain coupling, the acronym BJ is used to denote a block Jacobi-type iteration, in which the contributions from other subdomains are neglected. BGS1, BGS2, and BGS3 all indicate that blocks are implicitly coupled, the strength of which is determined by the trailing number. BGS1 iterations only update interface ΔQ after each subiteration; BGS2 iterations update after each subiteration and after each color change; and BGS3 iterations update after each subiteration, color change, and directional sweep. Note that for the Red-Black Jacobi algorithm (Section 5.4.1), BGS2 and BGS3 are identical, since there is no directional sweep involved. Likewise, for the symmetric Gauss-Seidel algorithm (Section 5.4.2), BGS1 and BGS2 are identical since there is only one node color involved. Figure 5.6 clarifies the four possible subdomain couplings.

Unfortunately, if the sequential memory leveraging hierarchy (Section 5.6.1) is used, the iteration hierarchy is limited to one update point at the beginning of each time step. Since no update is possible during linear subiterations, it is not possible to account for neighboring subdomain contributions during the relaxation algorithm. Thus, the matrix terms corresponding to nodes owned by other subdomains must simply be neglected. Unfortunately, as mentioned previously, it is this global communication of data during the subiterations that gives rise to the accelerated convergence rates enjoyed by implicit methods. Thus, the sequential iteration hierarchy forces one to use the BJ iteration, in which each subdomain is isolated from the rest during the matrix solution.

In summary, the handling of contributions from neighboring subdomains during the solution of the linear system strongly affects the convergence rate of the solver. One can choose to neglect the contributions (block Jacobi), or communicate these values at various points during the subiterative process. Obviously, stronger coupling between subdomains implies a higher overhead from message exchanges. The cost of message passing on the host architecture determines the optimal tradeoff such that total execution time is minimized.

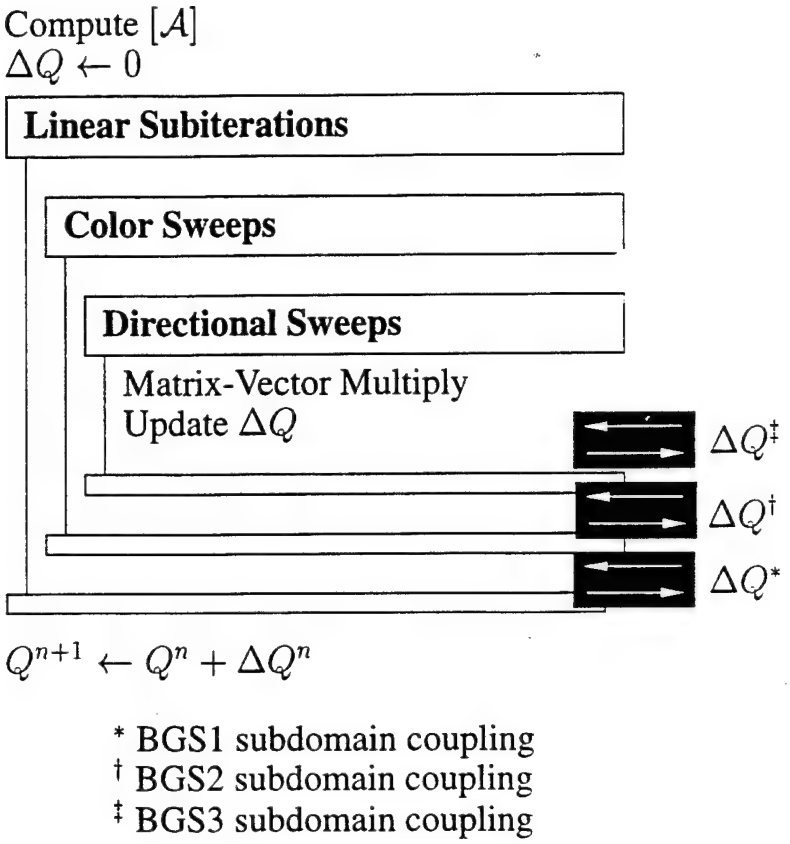


Figure 5.6: Definition of the BGS1, BGS2, and BGS3 subdomain coupling techniques; BJ iteration performs none of the above updates during the linear subiteration.

5.6.6 Domain Partitioning

The METIS software package [38] is used to partition the unstructured mesh. This package implements a set of multilevel graph partitioning algorithms [39], [40], [41] intended to perform efficient partitioning of arbitrary graphs. This implementation is very fast (a 1 million point grid can be partitioned in approximately one minute) as well as having the ability to minimize the number of cut edges (which decreases communication costs). Note that the partitions provided by METIS are not necessarily contiguous; however, that is not a problem for the current parallel solution algorithm.

CHAPTER VI

THE DEVELOPMENT OF AN OBJECT ORIENTED VISUALIZATION TOOLKIT

Although the purpose of this research was not explicitly focused on developing a visualization toolkit, the generation of a significant number of methods and algorithms warranted a closer look at properly packaging them for use beyond the scope of this problem. The following sections discuss the visualization paradigm used in this research, the general framework that encompasses the toolkit, and existing class definitions used for prototyping and developing the toolkit itself.

6.1 Visualization Paradigm for This Research

The work presented in this paper is limited to the analysis and development of the framework and the algorithms in a toolkit that operate on grids and solutions. This toolkit is implemented as a shared library that may be used in a broader scope for a variety of tasks. The manner in which the toolkit is currently used in the ERC is as a computational server in the broader context of a visualization package called DIVA (Data Interactive Visualization and Analysis). The system architecture of DIVA is illustrated in Figure 6.1

The box labeled Data Vault represents the disk storage needed to contain the input data. A Compute Server is shown as a separate entity that does CPU intensive calculations, for example: feature identification, isosurfaces, cutting planes, particle traces, etc. This Compute Server is the toolkit that contains the results from this work. The Graphics Server is the piece of the architecture that handles taking graphical primitives, such as triangle strips, and outputting them to either a file that contains a three-dimensional representation of the image, an image that can be generated or rendered off screen, or a computer that is capable of displaying interactive high-resolution displays. The connection between all of these pieces can be either high-speed network connections or can be resident memory. If the connection between the Data Vault and the

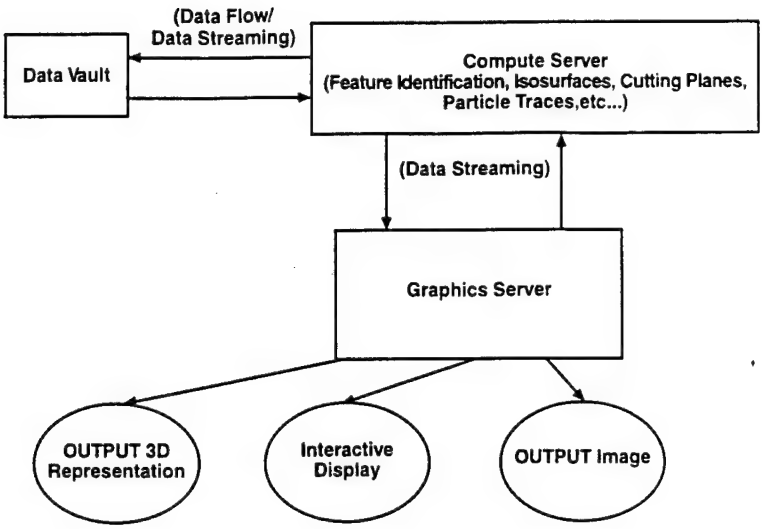


Figure 6.1: The system architecture for DIVA

Compute Server is a network, than the method of data transmission is one of either data flow or data streaming, depending on the type of the input data. Because we have imposed the restriction that an input data set need not change format, it is not possible to allow for data streaming of unstructured input data. Structured input data, however, may be streamed or decomposed into manageable units if necessary. If the connection between the Compute Server and the Graphics Server is a net ork, than the method of transmission is data streaming. The Compute Server acts as an extraction tool that takes a requested piece of input data, extracts the necessary information, and then passes it to the Graphics Server.

DIVA very closely matches the scientific visualization model as characterized by Springmeyer [42]. The visualization paradigm that DIVA embodies is one of focusing on the underlying process that is practiced by the user, one of designer-as-apprentice, discussed in Chapter 3. The paradigm present in DIVA also takes into account the data communication issues that have become a primary focus in all visualization packages currently in use. The issues that DIVA and the underlying toolkits face in terms of data management are similar to those introduced by Schroeder and Cox [43],[44], [45], in that this research has had to acknowledge and deal with data sets that are much too large for resident memory capacity.

6.2 The General Framework Encompassing the Toolkit

The Compute Server is implemented in C++ as a set of base classes. There also exists a set of C wrappers around these base classes that provide an external API to the toolkit. As was previously mentioned, the Compute Server is compiled into a shared library that can be called from an arbitrary front end. It is completely separate from the Graphics Server and has no specific ties to any graphics language. This toolkit serves simply to house a set of grids and solution, allowing for both queries and extractions on this data. This chapter does not discuss specific algorithms for either queries or extractions, but does serve to set the stage for how these algorithms come together to act as methods in this Compute Server. The Compute Server is capable of operating in a distributed fashion, meaning distributed from both the front end and from the graphics. It is designed to be able to communicate through pointers if all pieces reside on the same machine, either through a single processor, or through a shared memory arena. It is also designed to allow for distributed communication across a network through the Remote Procedure Calling (RPC) protocol using the External Data Representation (XDR) library that represents data structures in a machine-independent form [46]. RPC provides ability to communicate with procedures or processes outside of an application's current address space. This allows a local program to execute a procedure on a remote machine, passing data to it and receiving data from it. Using the RPC protocol, the Compute Server can reside on a geographically distant machine from a front-end graphical display. This makes the capability of the Compute Server much more attractive because this design directly addresses the concern of having to invest in expensive front-end graphical displays with extraordinary amounts of memory. Eventually, the size of the problem would prohibit having to have the Compute Server operate in the same address space as the graphical front-end. The Compute Server contains methods that allow for query capability, short burst questions that return reasonably small answers, and for extractions. Extractions are used to reduce the focus of interest from the entire data set to regions defined by a set of inputs. These extractions can be cutting planes, isosurfaces, particle traces, boundary surfaces, and in the most extreme sense, the volume itself.

Algorithms to perform both queries and extractions will be discussed in more detail in the next chapter. The following section provides an explanation of each of the classes that together make up the Compute Server.

6.3 Class Definitions

The classes are presented by discussing the rudimentary framework classes and administrative classes first. This gives an idea of how the entire Compute Server is put together to form a library of routines that work together to achieve a common set of goals. Presented after that are the grid classes that house every type of grid that may be supported in this current implementation. Grid components classes are discussed to define how base level extractions are made from the grids themselves that are native to specified grid types. Grouping classes are shown for facilitating multiple grid capabilities. Function value classes explain how both scalar and vector data are handled. Extraction classes define what types of extractions are possible through the current Compute Server. Graphical properties classes provide an introduction to graphics entities that are more naturally housed within the confines of the Compute Server.

6.3.1 Framework and Administrative Classes

The top level management object in the Compute Server is called the *csObject*. It contains a pointer to a class called *csLinkedList* and a void pointer to a rendering object. Because the Compute Server is designed to be independent of a graphical language, and because it may operate as a query and extraction device that outputs data in a file format as well as graphical, the rendering object pointer may be empty. This allows the Compute Server to behave in a batch type format that requires no front end display and may be driven by a command line interface. This capability, however simple it may be, has been invaluable in generating visualizations for the large scale data sets that have come through the ERC. It is not feasible to expect the user to sit at a workstation and wait for extractions to take place as they are being displayed. Rather, a situation in which a batch extraction can be performed and written to disk for later display has been very useful in a production visualization setting. The *csLinkedList* class embodies

the concept of a linked list of character strings with corresponding void pointers to entities and their corresponding enumerated types. Each entity that is a member of the *csObject* manager must have a name associated with it. The names are assigned at creation, and serve as a unique identifier for each entity. The *csLinkedList* class handles the creation of the linked list, additions, deletions, and queries. The next version of the Compute Server will allow for a *csHashTable* class to handle the management of these named entities. It is currently encumbant upon the user to avoid name clashes. Future versions will provide a capability to detect and correct name clashes. The named entities that are currently supported in the Compute Server are groups, grids, scalar and vector functions, blocks, computational surfaces, boundary surfaces, cutting planes, particle traces, isosurfaces, color bars and axes.

6.3.2 Grid Classes

The basic entity in the Compute Server is the grid. The function of this set of classes is to operate on either the grids themselves or to operate on extractions that were created from these grids. For this reason, much care and thought was put into the design of the grid classes themselves. There is a basic *csGrid* class that houses a set of base functions that are common to all types of grids and a set of virtual functions that must be defined uniquely inside each specific type of grid. The *csGrid* contains a pointer to a *csObject*, the manager object. It also contains the name of the *csGrid*, the type, the dimension, bounding volume information, the number of points inside the grid, the number of unique elements that make up the grid, and various other internal management information. A *csGrid* can be typed as unstructured, structured, mixed element, scattered, or unknown. The dimension is 2D, 3D, or 4D. The implementations of 2D and 3D grids are done separately to avoid any unnecessary memory allocations. A pointer to a *csGroup* is also retained to allow for a multiple grid capability. This is useful when a larger grid is decomposed into smaller units for computational purposes, but must be brought back together for extractions or display.

6.3.2.1 Virtual Definitions

A set of virtual functions that each grid must facilitate are defined in the *csGrid* class explicitly. These virtual functions include the capability to get element information, element neighbor information, obtain an element that contains a point, create the necessary extractions for an initial display, calculate cutting planes, and create isosurfaces. These virtual functions allow for higher level methods to insulate the details on a specific grid type from the algorithm itself. For example, the fundamental algorithm for calculating particle traces is the same in the abstract sense. It is only specific to a grid type in traveling from one element to the next (getting the element neighbor), interpolating function values inside a particular element, etc. The presence of virtual functions inside the *csGrid* class allow the formation of a generic *csParticleTrace* class that is capable of calculating particle traces on a variety of valid grid types. The virtual functions inside the *csGrid* class allow the details for the differences in behavior of each grid type to be encapsulated inside the specific grid itself. The individual grid classes that are supported are structured curvilinear, unstructured tetrahedral, unstructured mixed element, and scattered data. For the sake of accomplishing goal number three, minimizing resident memory usage, each of the supported grid classes was implemented separately. The Field Encapsulation Library (FEL) presented in Chapter 2 is written as a set of object oriented classes as well. However, it distinguishes algorithmic behavior at the element level [47]. The Compute Server distinguishes algorithmic behavior at the grid level. Computationally, the smallest named entity is a grid, or some extraction that has been taken from a grid. This treatment of class management is preferable when implementing algorithms such as querying an element neighbor. At the grid level, it is possible to insulate most of the detail inside the specific grid implementation class itself, and allows for refinements to improve both speed and memory usage. A structured curvilinear grid does not require an explicit neighbor map inside blocks. However, between block boundaries, an explicit neighbor map is required to travel from block to block seamlessly. FAST uses implicit neighbor mapping for traversal inside a block, but stops at block boundaries unless an IBLANKED grid has been input that explicitly gives block to block mapping [48].

The structured curvilinear grid implemented inside the Compute Server calculates this explicit neighbor map for block to block traversal transparently to the user. In an unstructured grid, an explicit neighbor map is required to be stored. This low level detail is transparent to the user by providing these virtual functions in the abstract *csGrid* class.

6.3.3 Grid Components Classes

There are lower level entities that form regions of interest in a grid that are not necessarily extractions in the purest sense. For structured grids, these are blocks and computational surfaces. A structured grid is made up of blocks that speak in terms of i, j, and k indices. At a specified block, i, j, or k index, a computational surface can be seen. The Compute Server allows for these entities as separate classes and unless the user specifically queries information from them; the details of these classes are transparent to the user. They serve to encapsulate the user from detail that is not critical information. The language in an unstructured grid is very different. An unstructured volume grid is made up of surface grids that represent boundary conditions and the field itself. Again, this information is hidden from the user unless it is specifically queried. These entities are automatically generated when the *csGrid* is created. Blocks and computational surfaces are generated only if the volume grid is structured curvilinear, and boundary conditions and boundary surfaces are generated only if the volume grid is unstructured.

6.3.4 Grouping Classes

A *csDataGroup* class is available for grouping instances of data objects that represent grids, solutions, and extractions. These entities are in turn pieces that, when combined, form a large grid. The large superset is decomposed into smaller units to enable parallelizing the computations of the solution values at the grid points. When trying to visualize this entity, it can be input as separate grid and solution objects that must be grouped to provide a single representation to the user. The *csDataGroup* allows the melding of these separate data objects into a single group. When extractions are computed, they are done so on the grouped entity. The user does not carry the burden of having to manually regroup these entities.

6.3.5 Function Value Classes

Function values are stored in the Compute Server in one of two classes, the *csScalarFunction* class or the *csVectorFunction* class. The two classes are very similar in nature. They have the same set of methods. However, the *csScalarFunction* manages scalar variables, and the *csVectorFunction* manages the vector variables. Both classes store the function name, a pointer to the associated grid class, a pointer to or a copy of the actual data values themselves, depending on the type of communication, and extrema information. All functions are required to have the same number of point values as the grid that it is associated with. The Compute Server handles only functions that come in with a grid. The grid is assigned at creation and must be specified by the user. The Compute Server does checking to ensure that the number of points in the function match the number of points in the specified grid. This grid is also required to be created previous to the creation of function values.

6.3.6 Extraction Classes

There are currently three types of extractions that are fully implemented inside the Compute Server: cutting planes, isosurfaces, and particle traces. The details of the algorithms are discussed in the next chapter. This section serves to point out that the management of the specific properties of each of the extractions is done in the classes themselves. The actual traversal or interpolation that is performed within the context of these specific algorithms is handled by the grid on which the extraction is being calculated.

6.3.7 Graphical Properties Classes

Although the purpose of the Compute Server is not necessarily tied to a graphics system, and does not have to be used to do any post-processing, there is a need to keep some graphical information present in the Compute Server. This graphical information is an overlap with the function data. For example, a *csColorBar* class is provided to manage color bar information that can be used to create and obtain the display of entities using a user specified color map. The color map is managed like all other entities in the Compute Server, by name. Any number of

colors can be specified to make up the color map. Additionally, a *csAxis* class is also provided for the management and display of legends and axes on the computational data.

6.4 Summary

An overview of both the design and the implementation of the Compute Server has been given. It is now relevant to begin discussing how the classes and the structure of these classes inside this entity facilitate the rapid prototyping of efficient methods for both the query and extraction of information from the given input data. This efficiency, as will be discussed in the next chapter, refers to both efficient memory usage and efficient computational time performance.

CHAPTER VII

DESCRIPTION AND ANALYSIS OF THE KEY VISUALIZATION TOOLKIT ALGORITHMS

This chapter presents and provides analysis for the key algorithms that make up the Compute Server. These algorithms have been refined and tailored to achieve a balance between both minimizing resident memory and optimizing speed performance. The concepts behind these algorithms are not new. Surface extraction and visibility ordering of unstructured polyhedra are not new to the visualization community. However, these algorithms have been optimized for performance in a computational setting for the purposes of feature extraction and visualization of large scale unstructured scientific data sets. These uniquely optimized algorithms placed in the framework discussed in the previous chapter make this work an original effort to fine tune the basic set of algorithms needed to compute and display features and extractions from large scale CFD data sets. This effort has certainly been a creative endeavor to investigate the inner workings of all components of the Compute Server. These components include the core data structures, search algorithms that use these data structures, and extractions that use both. This work was conducted as a result of investigating the current state-of-the-art and realizing that many of the existing techniques do not clearly delineate where the cost/performance line is. This is because many existing systems attempt to create a general framework and a general set of algorithms that is capable of handling any type of input data. The algorithms presented in this chapter have not thoroughly been investigated on all possible types of input data; rather, the scope has been limited to include only data from large scale unstructured CFD simulations.

7.1 Data Structures

Although data structures are a vital component in any algorithm, they are often the most overlooked component. An inappropriate choice for basic data structures will often lead to either

continuous algorithmic reworking to overcome this, or as is often the case, a total rewrite of the software with different data structures that have been found to be more applicable to the problem at hand. Given the premise that a balance must be achieved between both minimizing resident memory and optimizing speed performance, it was clear that the data structures had to be simple enough to minimize any overhead needed for constructs such as pointers, structures, classes, etc. The following sections outline the data structures that serve as the basis for all searching and unstructured grid traversal.

7.1.1 Defined Data Types

The Compute Server operates on a set of predefined data types [49]. These data types are used throughout all of the algorithms. Although the algorithms for traversing and visualizing unstructured volumes can be extremely complex, the data structures need not be. In fact, to accomplish the goal of minimizing memory usage, all significantly sized data is placed in a one-dimensional array. The defined types shown in Table 7.1 show how these one-dimensional arrays can be type defined. Placing large amounts of data in one-dimensional arrays eliminates any overhead in pointer allocations. Additionally, the manner in which these items are typed facilitates ease-of-use and improved readability of the code. The data placed in these typed arrays may be accessed as if they were allocated for its respective dimension. For example, an item vertex at index i in an array of FLOAT_3D values can be accessed as vertex[i][0], vertex[i][1], and vertex[i][2]. If an array of FLOAT_3D values is allocated for N values, then the amount of memory used is exactly 3*N. There is no overhead for pointers and no need for double dereferencing.

7.1.2 Array Indexing

All of the algorithms that are presented in this work operate on a variety of type defined one-dimensional arrays. The manner in which they operate on these arrays has a common theme [50]. This theme will be discussed using the structures needed to compute the elements surrounding a point. This algorithm is presented in detail in the next section.

Table 7.1: Defined Data Types in the Compute Server

```
typedef  char     CHAR_21[21];
typedef  int      INT_1D;
typedef  int      INT_2D[2];
typedef  int      INT_3D[3];
typedef  int      INT_4D[4];
typedef  int      INT_5D[5];
typedef  int      INT_6D[6];
typedef  int      INT_7D[7];
typedef  int      INT_8D[8];
typedef  float    FLOAT_1D;
typedef  float    FLOAT_2D[2];
typedef  float    FLOAT_3D[3];
typedef  double   DOUBLE_1D;
typedef  double   DOUBLE_2D[2];
typedef  double   DOUBLE_3D[3];
```

The following lines of code show the arrays needed to compute and store the elements surrounding a point. tNESP is used as a temporary array needed to construct nESP. nESP is the actual storage needed to construct the eSP array. eSP contains the actual elements surrounding each given point.

```
INT_1D  *tNESP  = NULL;  // A temporary array for constructing nESP.
INT_1D  *nESP   = NULL;  // Contains information for number elements surrounding a point.
INT_1D  *eSP    = NULL;  // Contains the specific elements surrounding each point.
```

The first operation in the creation of the elements surrounding a point is to allocate both nESP and tNESP. tNESP is allocated after nESP because it will be immediately deallocated once nESP has been fully constructed. Each of these arrays is allocated to be of length numberOfNodes+1. Each index in both arrays is initialized to zero.

```
nESP = new INT_1D[numberOfNodes+1];     // Allocate memory for actual array.
tNESP = new INT_1D[numberOfNodes+1];    // Allocate memory for temporary array.
```

```
for (i = 0; i <= numberOfNodes; i++)          // For i cycles over all nodes in grid plus one.
{
    tNESP[i] = 0;                             // Initialize temporary array to contain zeros.
    nESP[i] = 0;                              // Initialize actual array to contain zeros.
}                                             // End cycle over all nodes in grid plus one.
```

A pass is then made over all of the elements to increment the values in tNESP. After this loop, the values in tNESP reflect the actual number of elements surrounding each point.

```
for (i = 0; i < numberOfElements; i++)        // Cycle over all elements.
{
    for (j = 0; j < numNodesInElement; j++)// Cycle over all nodes in element i.
        tNESP[element[i][j]]++;               // Increment tNESP[i].
}
```

Once the arrays have been allocated and initialized, and the number of elements surrounding each point in the input grid has been set, nESP is updated to act as an index into the eSP array.

```
for (i = 1; i <= numberOfNodes; i++)  // For i cycles over all nodes plus one starting at one.
    nESP[i] = nESP[i-1] + tNESP[i-1]; // Increment nESP[i] to indicate all elements
                                      // surrounding a point up to that point.
delete[] tNESP;                       // Deallocate the memory needed for this temporary variable.
```

The last pass is made to loop back over all nodes of every element in the input grid. During this pass, an entry is made for each element index into the eSP array. This operation adds a given element i to the list of elements surrounding the points that make up the construction of the eSP array.

```
eSP = new INT_1D[nESP[numberOfNodes]];    // Allocate memory for content.
for (i = 0; i < numberOfElements; i++)    // Cycle over all elements.
{
    for (j = 0; j < numNodesInElement; j++)    // Cycle over all nodes in element i.
    {
```

```
            pIndex = element[i][j];          // Dereference the point index.
            eIndex = nESP[pIndex];           // Dereference the element index.
            eSP[eIndex] = i;                 // Add the element i to eSP.
            nESP[pIndex]++;                  // Increment index nESP[pIndex].
        }
    }
```

An example of this set of operations is shown in Figure 7.1. Part A shows tNESP after the first pass. Each of the values in tNESP reflects the number of elements surrounding that point. tNESP[0] = 4 indicates that point 0 has 4 elements surrounding it. Part B is an example of what nESP contains after it is updated. At this point nESP[0] reflects that point 0 has a beginning index of 0 into the eSP array. nESP[1] = 4 indicates that point 1 has a beginning index of 4 into the eSP array. Part C illustrates the contents of both nESP and eSP after the construction phase. nESP[0] = 4 indicates the ending index into eSP for all elements surrounding point 0. Because we are at the first point in nESP, the beginning index into eSP is 0. The elements surrounding point 0 are contained in indices 0, 1, 2, and 3. This is consistent with the contents of eSP containing elements E1, E2, E3, and E4. nESP[1] = 7 indicates the ending index into eSP for all elements surrounding point 1. The beginning index into eSP for point 1 is nESP[0] = 4. Again, this is consistent with the elements E5, E6, and E7, starting at index 4 and ending at index 6.

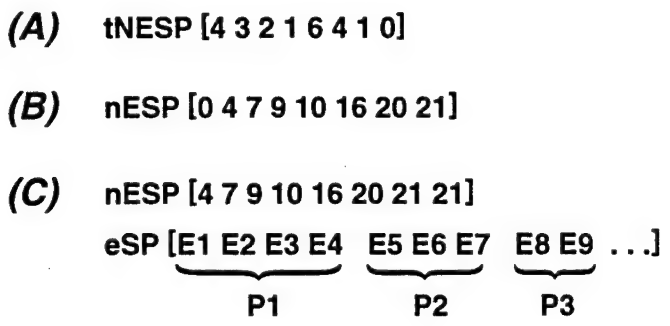


Figure 7.1: Index Arrays Used to Construct Elements Surrounding a Point

As was stated earlier, this method of using the one-dimensional arrays is one that is used frequently throughout all algorithms in the Compute Server. The next section presents a detailed algorithm and analysis for finding the elements surrounding each point in the input grid.

7.1.3 Elements Surrounding A Point

Before any of the actual algorithms can be performed, a set of base data structures must be created. The first is that of finding and recording the elements surrounding a point [50]. The concept of elements surrounding a point is illustrated in Figure 7.2. For a given point P, the set of elements surrounding this point are those whose faces contain the point P. In Figure 7.2, the elements surrounding point P are shown to be elements E1, E2, E3, E4, E5, and E6.

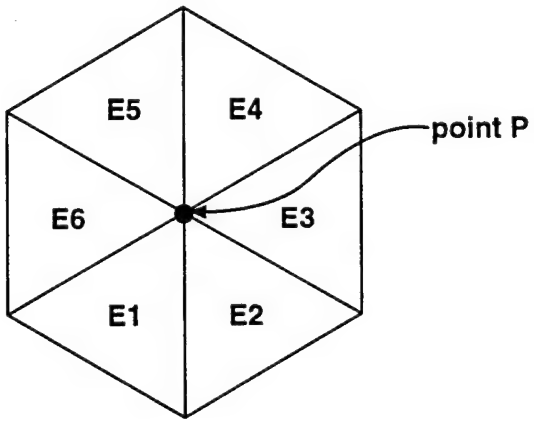


Figure 7.2: Elements Surrounding a Point

The following algorithm is based on that given in [50]:

1. Allocate the array nESP at a size of numberOfNodes+1. nESP is the array that will be used to both construct and index into the eSP array.
2. Allocate tNESP at a size of numberOfNodes+1. tNESP is a temporary array used to construct nESP.
3. Cycle over all nodes and initialize tNESP and nESP to zero.
4. Cycle over all elements in the input grid.

- Get the point index pIndex for each node of element i.
- Increment tNESP[pIndex] by one.

5. Cycle over all nodes i in the input grid starting at one, and set nESP[i]=nESP[i-1]+tNESP[i-1].
6. Delete the memory allocated for tNESP.
7. Allocate the array eSP at a size of nESP[numberOfNodes].
8. Cycle over all elements i in the input grid.

    - Get the point index pIndex for each node j of element i.
    - Let eIndex = nESP[pIndex].
    - Set eSP[eIndex] = i.
    - Increment nESP[pIndex] by one.

Both nESP and tNESP in steps 1 and 2 are allocated to be of length numberOfNodes+1, and are thus $\mathcal{O}(M+1)$ in memory where M is the number of nodes in the input grid. Step 3 of the algorithm cycles over all nodes and thus operates in $\mathcal{O}(M)$ time. The loop in step 4 cycles over all nodes in all elements in the input grid making it $\mathcal{O}(aN)$ in time, where N is the number of elements in the input grid, and a is a constant that represents the average number of nodes per element. Because we only deal with elements whose number of nodes range from four for tetrahedra to eight for hexahedra, the constant a is very small compared to N, and is in the worst case 8. Step 5 of the algorithm operates in $\mathcal{O}(M+1)$ time. Step 6 is the point in the algorithm where the temporary memory allocated for tNESP can be released. Because tNESP was allocated after nESP and no additional memory has been allocated, this release helps to avoid memory fragmentation issues. Step 7 allocates an array of size nESP[numberOfNodes]. Although we do not know the exact value of nESP[numberOfNodes] *a priori*, we can easily make a reasonable estimate for the types of input grids in this research. For a purely tetrahedral input grid generated for the purpose of solving computational fluid dynamics phenomena, the estimate would be $24 * M$, and is typically smaller than this. This estimate is a result of the manner in which the grid must be constructed to ensure reasonable quality. For a grid constructed solely of six noded pentahedra (prisms), the estimate would be $12 * M$. A purely hexahedral grid would

give an estimate of $8 * M$. Again, M is the number of nodes in the input grid. Given the above stated numbers, a worst case estimate of the amount of memory allocated for eSP is $\mathcal{O}(24 * M)$. An average estimate is $\mathcal{O}(16 * M)$. We can easily state this as $\mathcal{O}(cM)$ where c ranges from 8 to 24 for a mixed element input grid, and M is the number of nodes in the input grid. Step 8 operates in $\mathcal{O}(N)$ time where N is the number of elements in the input grid. An overall analysis of this algorithm tells us that it operates linearly in both space and time. It is $\mathcal{O}(cM)$ in space and $\mathcal{O}(aN)$ in time, $c << M$ and $a << N$. The detailed implementation of this algorithm is given in Appendix C.

7.1.4 Localized Decomposition Into Tetrahedra

The algorithms that are to be presented in future sections will operate on a variety of element types. As has been previously stated, these elements are tetrahedra, pyramids, prisms, and hexahedra. The basic type of element that all of the algorithms in the Compute Server operate on is the tetrahedra. All other types of elements that have been introduced can be locally decomposed into tetrahedra, and then each tetrahedra can be handled similarly. It is termed local since the decomposition is performed within the confines of the algorithm. No additional memory is required, and the operation is just a matter of indexing appropriately into the points making up the global element being decomposed. An example of the decomposition of a pyramid is illustrated in Figure 7.3. Part (A) shows the original pyramid. Part (B) shows how the pyramid can be decomposed to form two full face matching tetrahedra. The first tetrahedra is shown as being constructed by the points P1, P4, P3, and P5. The second tetrahedra is shown as being constructed by the points P1, P2, P3, and P4. Additionally, Part (C) gives an exploded view of the two tetrahedra that are combined to form a pyramid.

The decomposition of a prism is shown in Figure 7.4. The original prism is shown in part (A). Part (B) illustrates the combined view of the three tetrahedra that are generated as a result of the decomposition. They are full face matching. The first tetrahedra is constructed from the points P1, P2, P3, and P4. The second tetrahedra is constructed from the points P2, P3, P4, and

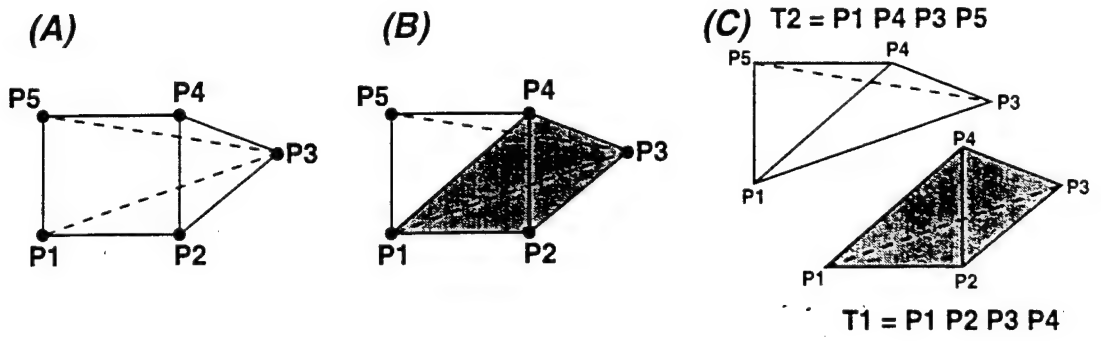


Figure 7.3: Local Decomposition of a Pyramid into Tetrahedra

P5. The third tetrahedra is constructed from the points P3, P4, P5, and P6. Part (C) gives an exploded view of the three tetrahedra.

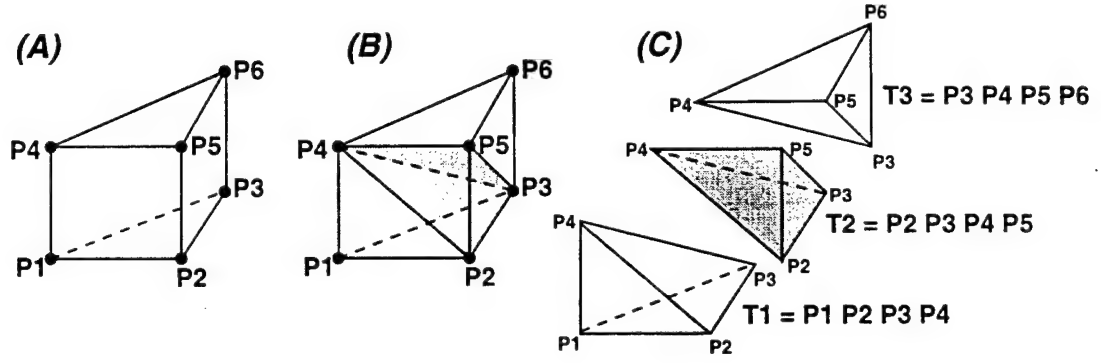


Figure 7.4: Local Decomposition of a Prism into Tetrahedra

The decomposition of a hexahedra is shown in Figure 7.5. The original hexahedra is shown in part (B). It is made up of 8 point indices P1, P2, P3, P4, P5, P6, P7, and P8. A hexahedra can be thought of as being divided into two prisms which can be further decomposed into three tetrahedra each. Part (A) illustrates the three tetrahedra that are a result of subdividing the first prism. The first tetrahedra is made up of points P1, P3, P4, and P7. The second tetrahedra is made up of points P1, P7, P4, and P8. The third tetrahedra is made up of points P1, P7, P8, and P5. Part (C) illustrates the three tetrahedra that are a result of subdividing the second prism. The first tetrahedra is made up of points P1, P2, P3, and P5. The second tetrahedra is made up of points P2, P3, P5, and P6. The third tetrahedra is made up of points P3, P5, P6, and P7.

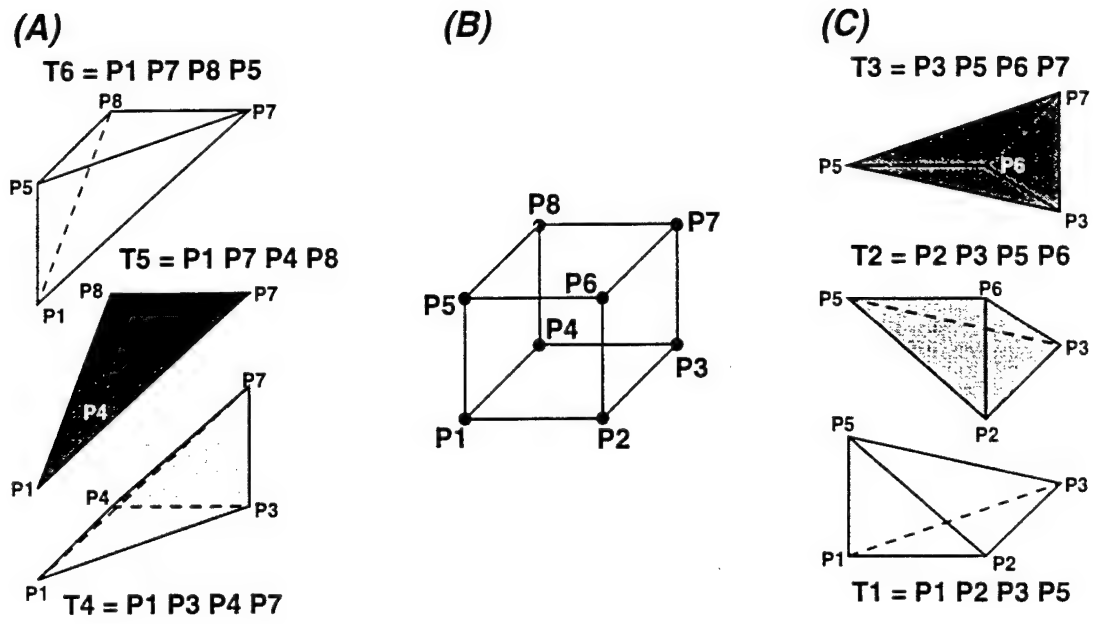


Figure 7.5: Local Decomposition of a Hexahedra into Tetrahedra

7.1.5 Element Neighbors

To conclude the presentation of the basic data structures, the creation of an element neighbor map is presented [50]. Before discussing the specific algorithm, the concept of an element neighbor is introduced. Figure 7.6 displays the element neighbors for a tetrahedral element. Element neighbor one, N1, is the element that contains a face that matches points P1, P2, and P3. Element neighbor two, N2, is the element that contains a face that matches points P2, P3, and P4. Element neighbor three, N3, is the element that contains a face that matches points P3, P4 and P1. Element neighbor four, N4, is the element that contains a face that matches points P4, P1 and P2.

Figure 7.7 displays the element neighbors for a pyramid. Element neighbor one, N1, is the element that contains a face that matches points P1, P2, and P3. Element neighbor two, N2, is the element that contains a face that matches points P2, P3, and P4. Element neighbor three, N3, is the element that contains a face that matches points P3, P4 and P5. Element neighbor four,

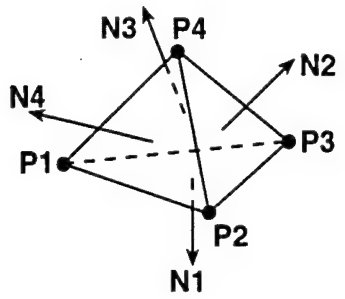


Figure 7.6: Element Neighbors For a Tetrahedra

N4, is the element that contains a face that matches points P1, P3 and P5. Element neighbor five, N5, is the element that contains a face that matches points P1, P2, P4, and P5.

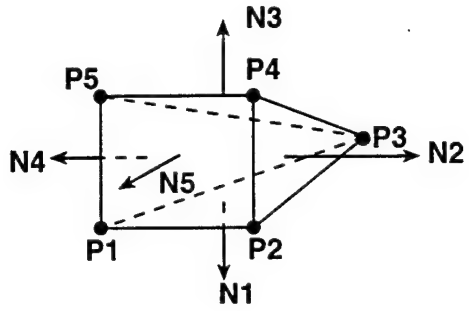


Figure 7.7: Element Neighbors For a Pyramid

Figure 7.8 displays the element neighbors for a prism. Element neighbor one, N1, is the element that contains a face that matches points P1, P2, and P3. Element neighbor two, N2, is the element that contains a face that matches points P2, P3, P6, and P5. Element neighbor three, N3, is the element that contains a face that matches points P4, P5 and P6. Element neighbor four, N4, is the element that contains a face that matches points P1, P3, P6 and P4. Element neighbor five, N5, is the element that contains a face that matches points P1, P2, P5, and P4.

Figure 7.9 displays the element neighbors for a hexahedral element. Element neighbor one, N1, is the element that contains a face that matches points P1, P2, P3, and P4. Element neighbor two, N2, is the element that contains a face that matches points P2, P3, P7, and P6. Element neighbor three, N3, is the element that contains a face that matches points P5, P6 P7, and P8.

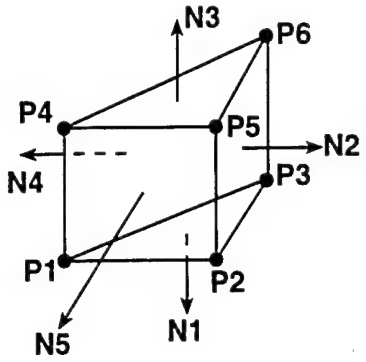


Figure 7.8: Element Neighbors For a Prism

Element neighbor four, N4, is the element that contains a face that matches points P1, P4, P8 and P5. Element neighbor five, N5, is the element that contains a face that matches points P3, P4, P8, and P7. Element neighbor six, N6, is the element that contains a face that matches points P1, P2, P6, and P5.

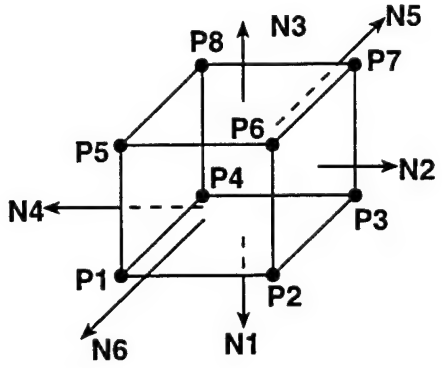


Figure 7.9: Element Neighbors For a Hexahedra

The algorithm to construct the element neighbor map is based on the algorithm that was presented in [50]. It assumes that the map containing the elements surrounding a point has already been constructed.

1. Allocate the array eN that will contain the element neighbor information.
   eN = new INT_6D[numElements].
2. Cycle over all neighbors j of all elements i and initialize each entry in eN.
   eN[i][j] = -555.

3. Cycle over all neighbors j for all elements i in the input grid.
   If eN[i][j] == -555 (the element neighbor has not been set),

   (a) Find the element ne and the face nf that matches element i at face j.

   (b) Set eN[i][j] = ne.

   (c) if ne != -999, eN[ne][nf] = i. (If ne is -999, then the neighbor for element i through face j is a boundary face. Otherwise, this value is a valid element neighbor.)

Step 1 of the algorithm shows us that the amount of memory required to construct the element neighbor map is $\mathcal{O}(6 * N)$ where N is the number of elements in the input grid. This version of the algorithm assumes that we are dealing with the mixed element type grid. One array is used to house the entire element neighbor map in a mixed element grid. It is sized according to the element with the most number of neighbors, the hexahedra. If the input grid is purely tetrahedral, then the size of the memory is actually $\mathcal{O}(4 * N)$, resulting in no wasted memory. There is only a waste of memory when dealing with mixed element grids. This can easily be corrected by keeping track of separate structures for tetrahedra, pyramids, prisms, and hexahedra. For demonstration purposes, we will keep a single structure and accept the memory overhead. Step 2 cycles over all neighbors of all elements and initializes all entries in the element neighbor structure. This loop operates in $\mathcal{O}(6 * N)$ in the worst case. Step 3 again cycles over all neighbors in all elements. If the element neighbor has not been set, then the first operation is to find the element containing a face that matches the current element and face at the nodes. Finding this common element operates in $\mathcal{O}(c)$ time where c is the maximum number of elements surrounding the nodes making up the face in question. As was stated in the section discussing elements surrounding a point, a reasonable worst case estimate for c is approximately 24. An average value would be somewhere in the range 8 to 24, making c very small in comparison to the number of nodes in the input grid. The other operations in the loop in step 3 operate in constant time, making the loop in step 3 operate in a total time of $\mathcal{O}(cN)$. Constructing the element neighbor map is performed exactly once for a given input grid. As long

as the grid connectivity does not change, the element neighbor map is valid. The algorithms that are presented in future sections require the use of this element neighbor map. After the element neighbor map is constructed, the memory needed for elements surrounding a point is deallocated. None of the algorithms in the Compute Server require explicit use of the elements surrounding a point. Thus, at this point, the required memory returns to $\mathcal{O}(6 * N)$ where N is the total number of elements in the input grid. A full implementation of the construction of the element neighbor map is shown in Appendix D.

7.2 Searching

Equal to the concern for the construction and memory usage needed for the base data structures is the concern for traversing the unstructured grid. Almost every algorithm in the Compute Server that performs a significant task requires some sort of searching to be done. The discussion of searching is shown in two parts. First, a presentation of a volumetric chunking algorithm is presented and analyzed. Second, the actual searching schemes which use this volumetric chunking are presented and analyzed.

7.2.1 Volume Chunking

Because searching is such a significant part of the navigation and display of unstructured volumes, a great deal of time was spent investigating existing methods for searching unstructured grids, and for possible new ideas to reduce both memory overhead and search time. Additionally, a significant problem occurs when searching through a volume that contains embedded boundaries. Using traditional searching techniques for unstructured grids [50], the search can hit an embedded boundary during the traversal and exit out of the search, forcing a global search of all elements in the grid. This situation is illustrated in Figure 7.10. The starting element is shown as being located in front of the embedded boundary; the actual element containing the given point is located behind the embedded boundary. Traditional searching techniques navigate the search in the direction vector from the starting element towards the given point. At some point in the search, the traversal collides with the embedded boundary, and fail; this forces a

brute force search of all unvisited elements. This is obviously an undesirable situation. For this reason, much work has been done to impose an additional structure on the existing data. Current methods include, but are not limited to, the creation of additional structures such as octrees, range trees, interval trees, etc. These structures are generally used to house or categorize the original data.

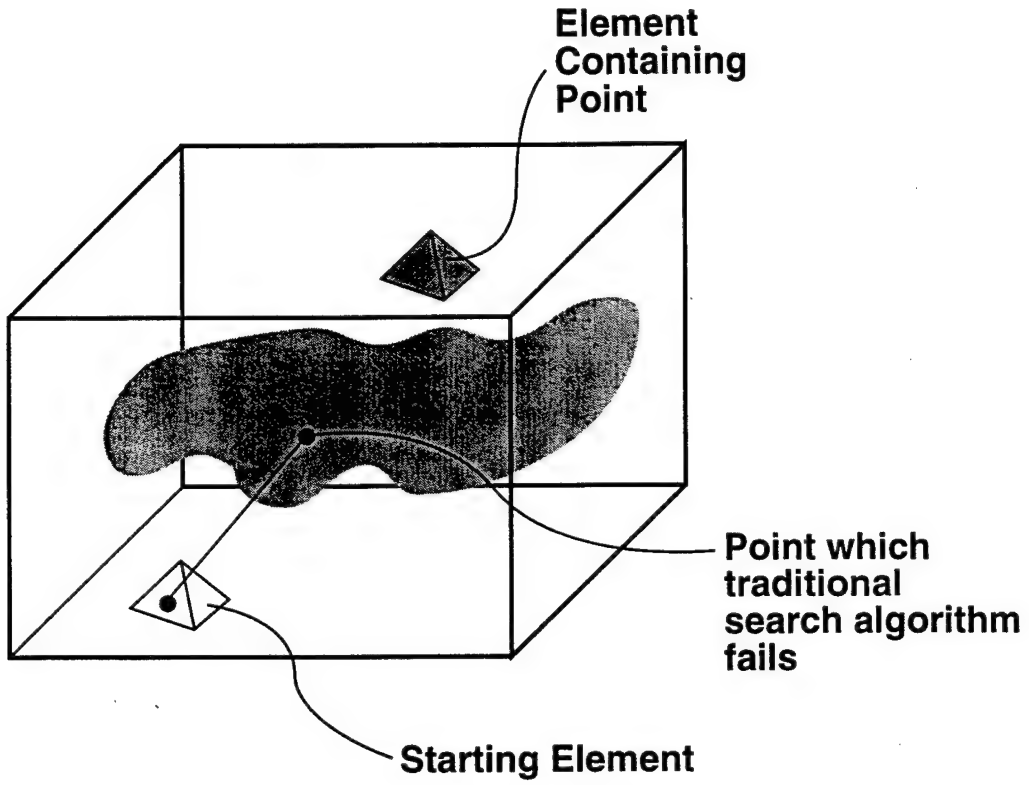


Figure 7.10: A Common Problem With Embedded Boundaries and Traditional Search Techniques

Methods that use octrees are perhaps the simplest to create and understand, and several versions of building and traversing octrees exist today [51], [52], [53]. Wilhelms and Van Gelder [54] use a branch-on-need octree to purge subvolumes in the creation of isosurfaces. This method has a worst case time efficiency of $\mathcal{O}(k + k\log(n/k))$ where n is the total number of cells, and k is the number of active cells [55]. This method applies only to structured data sets, and requires

significant changes to allow for unstructured input data. Octrees have been primarily applied to structured grids and are not easily adapted to deal with unstructured grids [55].

Range-based methods apply to both structured and unstructured data, but are generally more suitable for unstructured data because they are unable to exploit implicit adjacency information found in structured data sets. Range-based methods have higher memory requirements. Gallagher [56] proposes a method based on a subdivision of the range domain into buckets, and on a classification of intervals based on the buckets they intersect. The tradeoff between efficiency and memory requirements is highly dependent on the resolution of the bucketing structure [57]. Giles and Haimes [58] report an approach in which two sorted lists of intervals are constructed in a preprocessing phase by sorting the cells according to some predefined minimum and maximum function value. This method exploits the concept of global coherence. More recently, Shen and Johnson [59] try to improve and overcome some of the limitations presented in [56] and [58] by adopting similar structures to address global coherence. However, a worst case computational complexity of $\mathcal{O}(N)$ has been estimated for all range-based methods discussed above [55].

Livnat [55] introduced the concept of a span space, a two-dimensional space where each point corresponds to an interval in the range domain. The span space is useful to geometrically understand range-based methods. A kd-tree is used to locate the active intervals in this space, achieving an $\mathcal{O}(\sqrt{n} + k)$ time complexity in the worst case. In a more recent paper, Shen [60] proposed the use of a uniform grid to locate the active intervals in the span space. The overhead memory required to impose this kd-tree is approximately 25% above the memory required to house the original grid itself, and in many case greater than 25%.

During the course of this research, both the octree method and the range tree method were implemented and tested to determine usability. It was found that the memory overhead required for both the octree and the range tree outweighed any potential gain in both a global and local searching situation. The branch-on-need octree given by Wilhelms and van Gelder stated an overhead of approximately 20%. The kd-tree imposed on the original grid that was presented

above showed an overhead of over 25% above the memory required for the input data. For small problems, this overhead is not a significant problem. However, for large data sets such as those in this work, an overhead of greater than 20% can mean the difference between being able to visualize or analyze the problem and not being able to. For this reason, a simpler data structure was adopted to impose additional structure information on the input grid, while maintaining a memory overhead of less than 20%. This data structure and its use is termed "volume chunking". Given a number of x divisions, y divisions, and z divisions, a volume can be decomposed into subvolumes or chunks. A very simple version of a volume chunking scheme applied to a cubic volume is shown in Figure 7.11. In this example, the number of x divisions, y divisions and z divisions is equal to two.

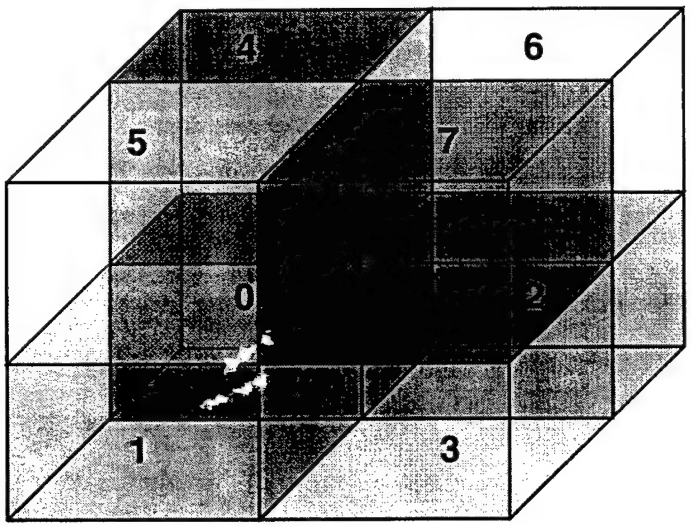


Figure 7.11: A Cube Chunked Into Eight Subvolumes

The algorithm for creating the volume chunks is given as:

1. Compute the bounding box.
2. Find x, y, and z increments based on the number of x divisions, the number of y divisions, and the number of z divisions. These values can be user supplied or can be computed by decomposing the volume based on a desired number of elements in each volume. Then numVolumes = x divisions * y divisions * z divisions.

3. Allocate the structure numElementsInVolume.
   numElementsInVolume = new INT_1D[numVolumes].
4. Allocate temporary tNumElementsInVolume for construction of numElementsInVolume.
   tNumElementsInVolume = new INT_1D[numVolumes].
5. Cycle over all volumes and initialize tNumElementsInVolume.
6. Cycle over all elements i in the input grid.
   (a) Calculate the volume index (vIndex) for the first node.
   (b) Increment tNumElementsInVolume[vIndex] by one.
7. Cycle over all volumes i and set: numElementsInVolume[i] = numElementsInVolume[i-1]+tNumElementsInVolume[i-1].
8. Delete the memory for structure tNumElementsInVolume.
9. Allocate elementsInVolume.
   elementsInVolume = new INT_1D[numElements].
10. Cycle over all elements i in the input grid.
    (a) Calculate the volume index, vIndex, for the first node of the element i.
    (b) Set elementsInVolume[numElementsInVolume[vIndex]] = i.
    (c) Increment numElementsInVolume[vIndex] by one.

The method of volume chunking uses the same array indexing scheme as that shown in the construction of elements surrounding a point. Steps 3 and 4 of the algorithm shown above allocate the amount of memory needed to construct the volume chunking array. This memory is seen to be $\mathcal{O}(V+1)$ where V is the number of subvolumes or volume chunks. Step 5 cycles over all of these volumes in $\mathcal{O}(V)$ time. Step 6 operates in order $\mathcal{O}(N)$ time where N is the number of elements in the input grid. Step 7 operates in $\mathcal{O}(V)$ time. Step 9 allocates the memory needed for the volume chunking structure $\mathcal{O}(N)$ space. Step 10 operates in $\mathcal{O}(N)$ time. This results in an overall space usage of $\mathcal{O}(N+V)$ and a construction time of $\mathcal{O}(N)$. Given the space requirements for the original grid, the additional space required to build the volume chunking structure is an overhead of approximately 18%. This overhead falls well below any that have been mentioned in the schemes presented above. The volume chunking scheme easily handles the problem of getting lost during the traversal. The searching scheme in the next section uses

this volume chunking to select a good starting element to begin the search. If the traversal gets lost during the search, the routine cycles over all elements in a given volume. The volume chunking structure is specifically used to obtain a good candidate starting element in a search if one is not already available. It is also used to reduce the number of possible elements to search through should an embedded boundary be contained in a given volume chunk. A discussion and analysis of this volume chunking method for searching is presented in the next section. An implementation of the volume chunking is given in Appendix E.

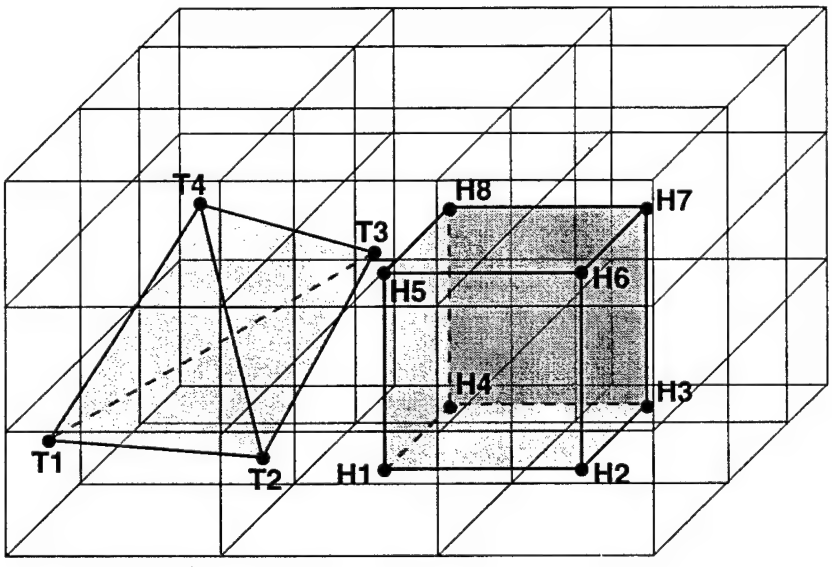


Figure 7.12: An Example of Volume Chunking

The process of finding the chunk that a given point is located in is a direct computation. Figure 7.12 illustrates the chunking of space that contains both a tetrahedra and a hexahedra. To find the chunk that the node T1 belongs to, a calculation is made to determine the x index, the y index, the z index, and the volume index. They are computed in the following manner:

```
getBoundingBox(Min,Max);
xIndex = (int)(((x-Min[0])/(Max[0]-Min[0]))*numXDivisions);
yIndex = (int)(((y-Min[1])/(Max[1]-Min[1]))*numYDivisions);
zIndex = (int)(((z-Min[2])/(Max[2]-Min[2]))*numZDivisions);
if (xIndex == numXDivisions)
    xIndex--;
if (yIndex == numYDivisions)
    yIndex--;
if (zIndex == numZDivisions)
    zIndex--;
vIndex = xIndex*numYDivisions*numZDivisions +
         yIndex*numZDivisions +
         zIndex;
```

7.2.2 Global and Local Searching Techniques

Searching is used for a variety of reasons during the process of visualizing scientific data, but the primary reason for searching inside the Compute Server is to determine element containment for a given point. The searching algorithm inside the Compute Server operates in two basic modes, global searching and local searching. The global searching technique uses the volume chunking data structure and the given point to locate a good starting element. Then the local search is invoked. If the local search is unable to find the element containing a given point, then all elements within the chunk are tested to see if the point is contained within any element in the grid. Although this is an exhaustive search of the volume chunk, the number of elements inside that volume chunk is much smaller than the number of elements in the input grid. There are only two reasons why the local search would fail to find an element containing a given point: (1) the given point is outside of the volume or inside a cavity that contains no elements and (2) there

exists an embedded boundary in the volume chunk that was chosen to contain the give point. Presented next is the method for using volume coordinates to determine search direction and element containment, the base searching algorithm, and a recursive local searching algorithm.

7.2.2.1 Using Volume Coordinates To Determine Point Containment

The calculation of volume coordinates is used to determine point containment and the sub volumes that are calculated are used to determine direction of traversal. This method is based on that given in [50]. A full implementation is given is Appendix F. Figure 7.13 illustrates how the computations of the subvolume V1, V2, V3, and V4 guide the traversal through the unstructured grid. V1 is calculated as the volume of the tetrahedra formed by the given point, P2, P3, and P4. If this volume is positive, then the given point is on the inside of the face formed by P2, P3, and P4. If it is negative, then the given point is on the outside of the face formed by P2, P3, and P4. If it is zero, then the given point lies on the face formed by P2, P3, and P4. V2 is calculated as the volume of the tetrahedra formed by the given point, P1, P4, and P3. If V2 is positive, then the given point lies on the inside of the face formed by P1, P4, and P3. If V2 is negative, then the given point lies on the outside of the face formed by P1, P4, and P3. Similar conditions apply to the computation of subvolumes V3, and V4.

7.2.2.2 Searching Algorithm

The base searching algorithm can be stated in the following steps, and is based on the searching algorithm given in [50]:

1. If a good starting element is needed, find one using the volume chunking structure.
2. Recursively search for element containment.
3. If the recursive search fails to find an element, then cycle over all elements in the volume chunk containing the given point that have not previously been visited.

The recursive algorithm is given the point, the current element in the traversal, an array of flags indicating whether an element has been visited, and the current visit. This algorithm can be stated as:

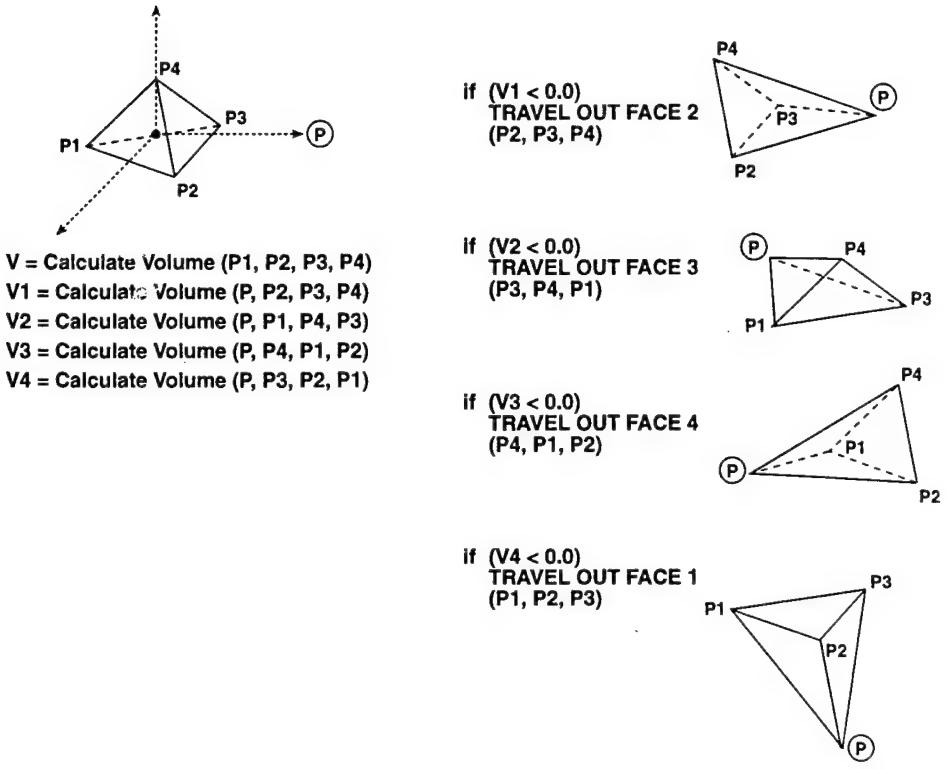


Figure 7.13: Volume Coordinates

1. Set the visited flag for the current element e to the current visit.
2. Calculate whether the current element contains the given point by computing subvolumes V1, V2, V3, and V4.
3. If $V1 > 0$ and $V2 > 0$ and $V3 > 0$ and $V4 > 0$, then the current element contains the given point. Return successfully with the current element.
4. If $V1 < 0$ or $V2 < 0$ or $V3 < 0$ or $V4 < 0$,
   - If the volume coordinate V1 is negative and element neighbor 2 is valid and not visited, then set the current element to element neighbor 2, and call the recursive search again.
   - If the volume coordinate V2 is negative and element neighbor 3 is valid and not visited, then set the current element to element neighbor 3, and call the recursive search again.
   - If the volume coordinate V3 is negative and element neighbor 4 is valid and not visited, then set the current element to element neighbor 4, and call the recursive search again.
   - If the volume coordinate V4 is negative and element neighbor 1 is valid and not visited, then set the current element to element neighbor 1, and call the recursive search again.

Finding an initial starting element using the volume chunking structure operates in $\mathcal{O}(1)$ time. Given a good starting element, the recursive search to find the containing element operates by navigating through the number of elements between the starting element and the containing element, provided no embedded boundary is encountered. This would operate in a worst case time of $\mathcal{O}(N^{1/3})$, where N is the number of elements in the input grid. This worst case estimate is based on a rectilinear three-dimensional volume that contains N elements. The diagonal through the volume is then given as $\mathcal{O}(N^{1/3})$. If the recursive search should fail, then the search routine would operate in a worst case time of $\mathcal{O}(nV)$, where nV is the number of elements contained in volume index V. Implementations of the searching algorithms are given in Appendix F.

As is shown in the algorithm above, an element containment test is performed. If the given point is not contained within the candidate element, then the resulting volume coordinates are used to determine the direction of traversal. Completing the discussion of the searching requires an explanation of the computation of volume coordinates. This computation calculates the sub volumes V1, V2, V3, and V4.

7.3 Cutting Planes

Cutting planes are perhaps the most common method for extracting information from a volume solution. Displayed properly, it can give a variety of meanings, from the shape and structure of the grid at that plane, to the behavior of the solution in that plane. It is a widely accepted method for querying the physical properties of a solution, however, it can also be one of the most costly. If the original volume grid and solution are significantly large, an arbitrarily placed cutting plane can potentially be very large as well. Because it is very widely used, and because it has the potential for displaying a great deal of information at one time, it is important to make sure that the cutting planes are generated optimally with respect to space and quickly with respect to CPU performance. The following sections present a new cutting plane algorithm that generates a cutting plane with no duplication of intersection points and by avoiding redundant element visits in the grid. As the cut is generated, the knowledge of where the

cut has already occurred is retained to avoid unnecessary intersection calculations and to avoid duplicating memory by storing redundant point information. This method can handle multiple disjoint grids, and it can handle all previously described element types. The cutting plane is specified by a single point and a single normal. The algorithm proceeds as follows:

1. Initialize all global structures and arrays needed to mark visits to elements.
2. Find an element e containing a point that lies in the cutting plane.
3. Intersect the plane with the element and record the intersection points and the triangles. At this point, we do not have to worry about duplicating points because the intersection routine returns a copy of the points listed exactly once. The intersection routine also returns information giving which element faces have been intersected.
4. Set this element e to visited.
5. Call the recursive cutting plane algorithm to grow the cutting plane out from element e.

The recursive cutting plane algorithm can be stated as:

1. If face 1 of e has been intersected and the neighbor attached to face 1 is valid and has not been visited, then
   (a) Calculate intersection parameters to determine whether element neighbor 1 intersects the plane. If the element intersects the plane,
   (b) Find the intersection points and intersecting faces of the plane with element neighbor 1.
   (c) Record all non-duplicate points.
   (d) Record all triangles.
   (e) Set element neighbor 1 to visited.
   (f) Call the recursive cutting plane algorithm to grow the cutting plane out from element neighbor 1.
2. If face 2 of e has been intersected and the neighbor attached to face 2 is valid and has not been visited, then
   (a) Calculate intersection parameters to determine whether element neighbor 2 intersects the plane. If the element intersects the plane,
   (b) Find the intersection points and intersecting faces of the plane with element neighbor 2.
   (c) Record all non-duplicate points.

(d) Record all triangles.

(e) Set element neighbor 2 to visited.

(f) Call the recursive cutting plane algorithm to grow the cutting plane out from element neighbor 2.

3. If face 3 of e has been intersected and the neighbor attached to face 3 is valid and has not been visited, then

   (a) Calculate intersection parameters to determine whether element neighbor 3 intersects the plane. If the element intersects the plane,

   (b) Find the intersection points and intersecting faces of the plane with element neighbor 3.

   (c) Record all non-duplicate points.

   (d) Record all triangles.

   (e) Set element neighbor 3 to visited.

   (f) Call the recursive cutting plane algorithm to grow the cutting plane out from element neighbor 3.

4. If face 4 of e has been intersected and the neighbor attached to face 4 is valid and has not been visited, then

   (a) Calculate intersection parameters to determine whether element neighbor 4 intersects the plane. If the element intersects the plane,

   (b) Find the intersection points and intersecting faces of the plane with element neighbor 4.

   (c) Record all non-duplicate points.

   (d) Record all triangles.

   (e) Set element neighbor 4 to visited.

   (f) Call the recursive cutting plane algorithm to grow the cutting plane out from element neighbor 4.

This cutting plane algorithm produces a plane that has a minimal representation of points given the resulting triangulation. The algorithm operates in worst case $\mathcal{O}(P)$ time, where P is the total number of elements in each volume chunk that is intersected by the cutting plane. An implementation of the cutting plane algorithm is given in Appendix G.

An example of how the algorithm travels from one face to the next is given in Figure 7.14. The tetrahedra, E1, formed from the points P1, P2, P3, and P4 shares a face with the tetrahedra, E2, formed from the points P1, P2, P3, and P5. E1 is intersected with the cutting plane resulting

in the intersection points I1, I2, I3, and I4. The intersection routine flags face 1 as having been cut and the traversal carries out through face 1 into element E2. E2 returns an additional intersection point of I5, recognizing the duplicates I1 and I2.

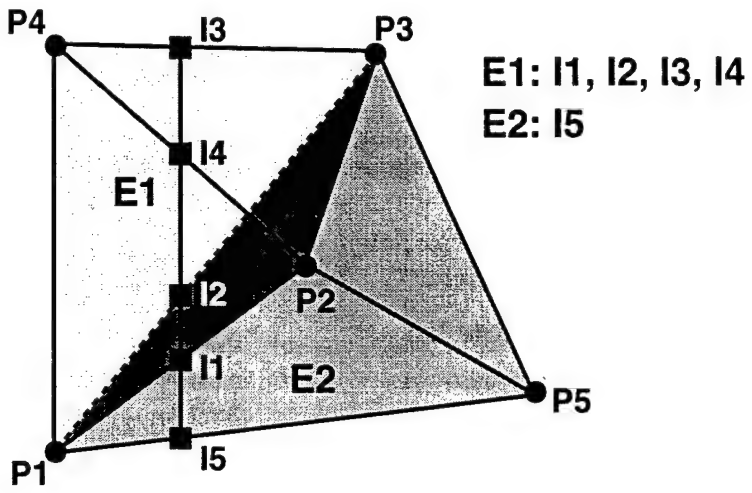


Figure 7.14: An Example of the Cutting of Neighboring Elements

7.4 Isosurfaces

Although this research is not specifically concerned with data structures for explicit isosurface creation, the methods used to create the cutting plane with no duplicate points can be extended to handle isosurface creation with no duplicate points. The creation of an isosurface is quite similar to the creation of a cutting plane. The volume chunking can also be extended to contains elements in chunks based on function values as opposed to spatial location. The memory overhead and timings are exactly the same as that for volume chunking and the cutting plane algorithms already presented.

7.5 Summary

A set of data structures and algorithms has been presented which operate well on large scale unstructured scientific data. These algorithms and data structures are implemented within the

confines of a framework called the Compute Server. It is now prudent to summarize the analysis of the new algorithms that have been presented and to provide performance examples.

A new method for searching through an unstructured volume has been presented. When the input data is brought into the system, an additional structure is imposed on this data called a volume chunking structure. This structure is used to avoid an exhaustive search of the volume when search methods fail due to the presence of embedded boundaries. The search method consults the volume chunking structure only when a good candidate element is needed and not already known. Once the volume chunking structure returns a good candidate element, the search navigates through the volume using a local search. If the local search fails due to an embedded boundary in the volume chunk in which the point is contained, the search will traverse brute force through all elements in the selected volume chunk. The overhead for this volume chunking structure is less than 18%. The overhead is computed by dividing the amount of memory needed for the volume chunking structure by the memory that is allocated for the input data. The worst case performance for a search is $\mathcal{O}(V)$, where V is the number of elements contained in the volume chunk selected. V is typically much less than the number of elements in the entire volume, thus improving significantly the exhaustive search needed when an embedded boundary is encountered. An average search time for the new method is $\mathcal{O}(N^{1/3})$. As a comparison, a brute force method for navigating through the volume results in a time estimate of $\mathcal{O}(N)$, where N is the number of elements in the volume. It has no overhead. A local area search with an element neighbor map, like that used in Field Encapsulation Library [47], has an average traversal of $\mathcal{O}(N^{1/3})$ and a worst case traversal time of $\mathcal{O}(N)$, with no overhead. The octrees presented by Wilhelms and van Gelder [54] have an impressive search time of $\mathcal{O}(K + K\log(N/K))$, where K is the number of active elements and N is the total number of elements in the volume. However, the overhead is approximately 20%, and the method is unable to handle embedded boundaries and unstructured data sets. A method using k-d trees by Livnat, et al. [55], has a search time of $\mathcal{O}(\sqrt{N} + K)$, where K is the number of active elements and N is the total number of elements in the volume. An extension to the k-d trees, interval trees,

presented by Cignoni, et al. [57], has a search time of $\mathcal{O}(\log N + K)$, where K is the number of active elements The overhead for both of these structures is $> 25\%$ and can be much larger than 25%. An example of three different size data sets is shown in Table 7.2. The Fighter is a swept wing fighter notional geometry from McDonnel Douglas. The X-38 is a geometry for the NASA escape pod for the international space station, and the Minita is a notional tilt-rotor geometry.

Table 7.2: Example Data Sets For Timing Algorithms

	Nodes	**Triangles**	**Tetrahedra**	**Pyramids**	**Prisms**
Fighter	64,924	24,322	349,018	0	0
X-38	335,274	41,786	1,943,483	0	0
Minita	4,806,397	462,872	7,439,997	34,517	6,875,063

Timing results for the modified searching algorithm discussed above are shown in Table 7.3. All timings were performed on a Silicon Graphics Octane MXE with a 300 MHz R12000 processor and 2 GBytes of main memory. Three separate methods for searching were timed. The method labeled Brute Force indicates an exhaustive search through the entire volume given an instance where an embedded boundary becomes an obstacle. The column labeled Traditional Searching reveals timings for a method that starts with element 0 and proceeds to navigate through the volume with local search techniques until an embedded boundary is encountered. The method then performs an exhaustive search through all elements in the volume that have not previously been visited. The column labeled Modified Searching shows the timings for the new searching algorithm that has been developed as a result of this research. These timings are encouraging. The modified searching technique appears to remain relatively constant in the amount of time it requires to find the desired element regardless of the size of the volume. Both brute force and traditional searching techniques show a significant increase in the amount of time required when the size of the data increases dramatically. The overhead also appears to become less significant as the size of the data grows.

Table 7.3: Timing Results For Modified Searching Algorithm (in CPU seconds)

	Brute Force Searching	**Traditional Searching**	**Modified Searching**	**Size of Grid**	**Size of Overhead**	**% Overhead**
Fighter	0.63	0.56	0.03	8.83 MB	1.40 MB	15.8
X-38	3.49	3.25	0.024	47.42 MB	7.78 MB	16.4
Minita	46.96	13.33	0.04	463.04 MB	57.40 MB	12.4

A new cutting plane algorithm has been presented that creates an arbitrary cutting plane with an optimal number of points given the default triangulation. This default triangulation refers to the set of triangles that result from cutting the elements without any compression techniques applied to the triangulation. The new method for calculating an arbitrary cutting plane operates in $\mathcal{O}(P)$ time, where P is the number of elements contained in each of the volume chunks that intersect the cutting plane. Given a judicious choice for the number of volume chunks, V, P can be seen to be $\mathcal{O}(N^{2/3})$. It can also be shown that eliminating duplicate points from the cutting plane will result in a maximum of $\mathcal{O}(6)$ compression. Because we are dealing with volumes generated for the purposes of computational fluid dynamics, we can be assured that there exists reasonable quality in the grids. This reasonable quality equates to an arbitrary cut that has an average of 6 elements surrounding each point. This is because the distributions of the angles in the surface grid are between 50° to 70°. An average angle of 60° results in 6 elements surrounding a given point. With an average of 6 elements surrounding each point, we can easily see that each point could be represented 6 times for each triangle. With elimination of duplicate points, we can see that a maximum compression of $\mathcal{O}(6)$ is achieved. FEPLOT3D [61] operates in $\mathcal{O}(N)$ time and has a duplicate point representation. The cutting plane algorithm in FAST [48] operates in a similar $\mathcal{O}(N)$ time with duplicate points. The Field Encapsulation Library [47] operates in $\mathcal{O}(N^{2/3})$ time with a duplication of points. Timing results for the modified cutting plane algorithm discussed above are shown in Table 7.4. These timings show a speed performance improvement of 3 to 4 times over traditional cutting plane algorithms. The traditional cutting plane algorithms refer to those that visit every element in the volume and do

not exploit coherency through the use of an element neighbor map. As expected through the complexity analysis, the compression is seen to be in the range of 5 to 6 times that of traditional methods. Again, these methods appear to scale well with the problem size.

Table 7.4: Timing Results For Modified Cutting Plane Algorithm (in CPU seconds)

	Traditional Cutting	**Modified Cutting**	**Speedup**	**Compression**
Fighter	0.68 25422 Points	0.18 5062 Points	3.78X	5.02X
X-38	3.59 117798 Points	0.87 21253 Points	4.13X	5.54X
Minita	29.09 353721 Points	6.96 66116 Points	4.18X	5.35X

Analysis of these data structures and algorithms has been presented, and a justification for the means of implementation has been discussed. The next chapter provides a summary of the information that has been presented in this research.

CHAPTER VIII

RESULTS AND CONCLUSIONS

8.1 Results

The results of this research are culminated in the development of a set of algorithms and the generation of an animation depicting the separation of a single booster from the delta II configuration. What follows is an overview of the animation procedures that were used to generate an unsteady animation of this event and an explanation of each segment of the resulting movie.

8.1.1 Animation Procedures

Each frame of the animation was generated through a batch version of the DIVA software and was rendered using a modified version of POVRAY [62]. The frames were generated on both a Sun Enterprise 10000 (E10000) with 64 processors each containing 2GB of main memory and a Sun Cluster with 64 processors each containing 2 GB of main memory. The segments were produced independent of each other and were run in parallel on the E10000 or the Cluster. The solution originally produced 1180 data sets, however the animation was generated by sampling every fourth data set resulting in 297 frames for the movie. The movie contains 4 segments of unsteady information and 3 segments of static information. The 4 unsteady segments contain 297 frames each, making the total animation contain 1191 frames. Each frame was ray traced on either the E10000 or the Cluster and required approximately 5 minutes - 1 hour to render. The unsteady segments were run in parallel on either the E10000 or the Sun Cluster using 32 processors at a time.

8.1.2 Storyboard for the movie

An initial storyboard was planned out to try to depict the movement and behavior of the tumbling booster over the 297 time steps. The final animation was divided into eight segments, and was designed to portray the behavior and the movement of the tumbling booster in the most informative manner. Each of the individual sections is described in the following paragraphs.

8.1.2.1 Full delta II viscous configuration

A viscous solution of the full delta II configuration was generated to serve as a frame of reference for the problem. Initially, the animation begins with a view of the delta II with contours of density from a viscous solution plotted on the body itself. The image that is in the movie is shown in Figure 8.1.

8.1.2.2 Close-Up of booster separating from full delta II viscous configuration

The viewer is then shown a close up of on the boosters that is beginning to separate from the full configuration. This image is shown in Figure 8.2.

8.1.2.3 Overall tumbling trajectory of booster separation

An overall view of the trajectory of the tumbling booster is shown in Figure 8.3. This is given to orient the viewer to the overall motion and path that the booster travels from separation to the end of the simulation. The initial frame is at the top of the figure and the final frame is given at the bottom right of the figure. The figure contains 99 of the frames that were used to generate the animation.

8.1.2.4 Animation of density contour on normal cut from rigid pole camera view point

The first unsteady sequence of the movie shows density contours on a cutting plane normal to the booster. During the sequence, each plane is calculated in the same location relative to the booster as it is tumbling. The camera is placed at an initial position some distance away in the z direction. During the animation, the camera remains attached to the body as if it were attached by a rigid pole. When the booster rolls, the camera rolls with it. Figure 8.4 shows the initial

frame of the animation at time t=0s. The density contours are plotted with a minimum value of 0.0 and a maximum value of 1.7. The image shown is Figure 8.5 shows the booster in this same animation sequence at time t=21.8s. The image shown is Figure 8.6 shows the booster in this same animation sequence at time t=29.5s.

8.1.2.5 Animation of density contour on normal cut from dynamic xy camera view point

The second unsteady sequence of the movie shows density contours on a cutting plane normal to the booster. During the sequence, each plane is calculated in the same location relative to the booster as it is tumbling. The camera is placed at an initial position some distance away in the z direction. During the animation, the camera remains moves in the x and y directions with the body, but remains fixed in the z. This means that the distance in x and y that the booster travels, the camera does also. However, if the booster rolls toward the camera (moves toward that camera), then the booster appears to get larger. If the booster rolls away from the camera, then the booster appears to get smaller. Figure 8.7 shows the initial frame of the animation at time t=0s. The density contours are plotted with a minimum value of 0.0 and a maximum value of 1.7. The image shown is Figure 8.8 shows the booster in this same animation sequence at time t=21.8s. The image shown is Figure 8.9 shows the booster in this same animation sequence at time t=29.5s.

8.1.2.6 Animation of density contours on two normal cut from dynamic xy camera view point

The third unsteady sequence of the movie shows density contours on two cutting planes normal to the booster. During the sequence, each plane is calculated in the same location relative to the booster as it is tumbling. The camera is placed at an initial position some distance away in the z direction. During the animation, the camera remains moves in the x and y directions with the body, but remains fixed in the z. The cuts are made opaque or transparent based on the dot product of the view vector with the normal to the cutting plane. The view vector is defined as the vector that is generated from the position that the camera is looking at minus the position on which the camera is physically located. The cuts are rendered completely transparent if the

normal to the cut is orthogonal to the view vector (the dot product of the normal to the cut and the view vector is 0.0). The cut is rendered as completely opaque if the dot product of the normal to the cut with the view vector is 1.0. A linear variation is used for any value between 0.0 and 1.0. This sequence allows the user to always have a full view of the solution parameters while maintaining the view of the full motion. The first unsteady sequence allowed for a full view of the solution, but did not allow for an accurate view of the full motion. The second sequence sacrificed the full view of the solution to allow the user to see an accurate view of the full motion. Figure 8.10 shows the initial frame of the animation at time t=0s. Again, the density contours are plotted with a minimum value of 0.0 and a maximum value of 1.7. The image shown is Figure 8.11 shows the booster in this same animation sequence at time t=21.8s. The image shown is Figure 8.12 shows the booster in this same animation sequence at time t=29.5s.

8.1.2.7 Animation of isosurfaces of two density values from dynamic xy camera view point

The final unsteady sequence of the movie shows time varying isosurfaces on the volume solution of the tumbling booster that grow and shrink over time. The isosurfaces are generated from density values at 0.45 and 1.35. The isosurface shown in magenta is density=1.35 and the isosurface shown in blue is density=0.45. The camera is placed at an initial position some distance away in the z direction. During the animation, the camera remains moves in the x and y directions with the body, but remains fixed in the z. Figure 8.13 shows the initial frame of the animation at time t=0s. The image shown is Figure 8.14 shows the booster in this same animation sequence at time t=21.8s. The image shown is Figure 8.15 shows the booster in this same animation sequence at time t=29.5s.

8.2 Conclusions

Visualizing the results from large scale unstructured unsteady simulations is an interesting and creative process. Several areas of research were brought together to develop both the algorithms and the resulting animation. These areas include grid generation for both static and

dynamically changing grids, parallel unsteady solution methods, and visualization techniques for extracting and rendering high quality animations.

Fortunately, the results of this research, were developed as a result of working closely in a multi-disciplinary team environment. This has enabled individuals from a variety of backgrounds to have not only theoretical input, but to have technical say in the outcome of the present work.

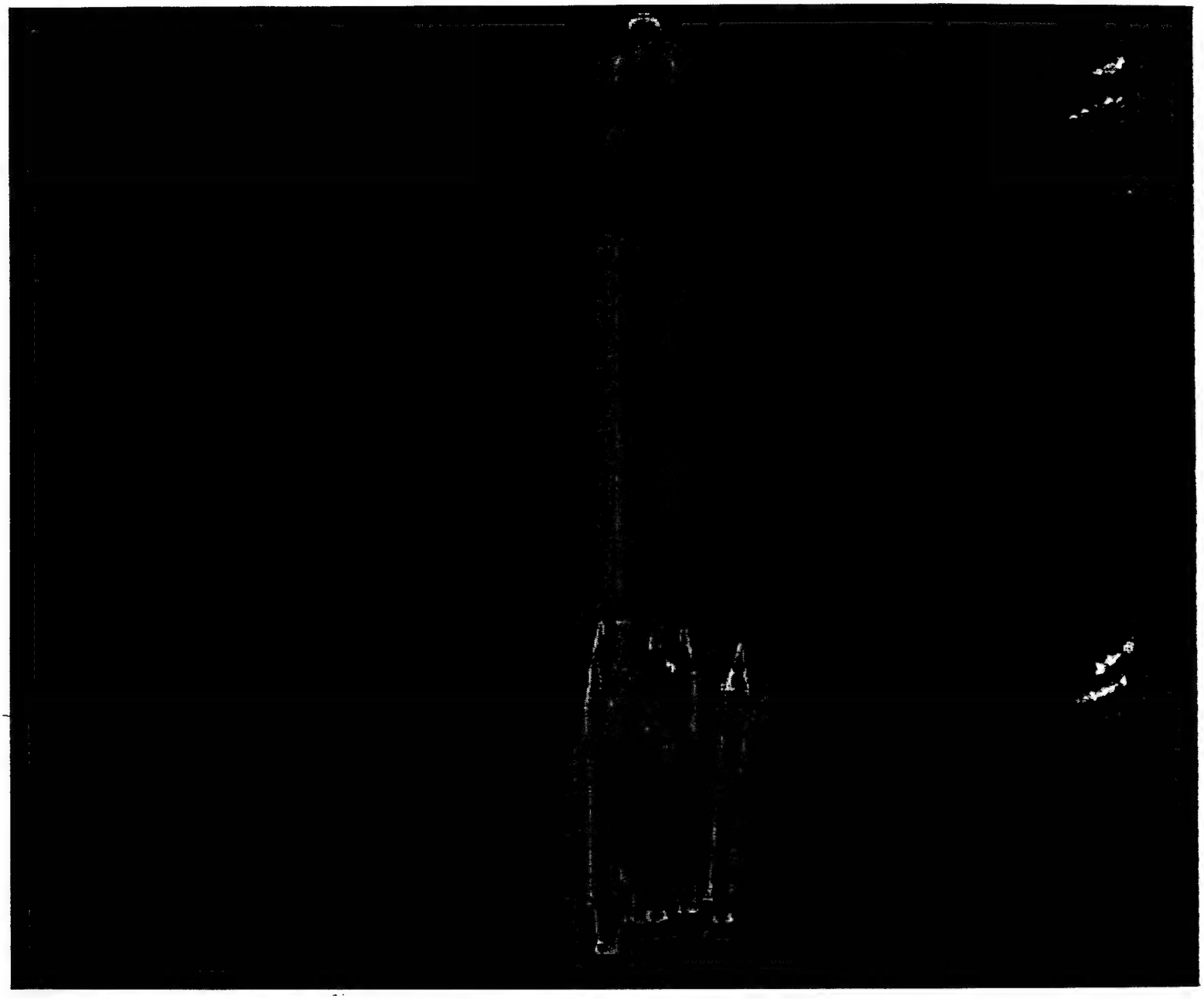


Figure 8.1: Viscous solution of the Delta II and the boosters

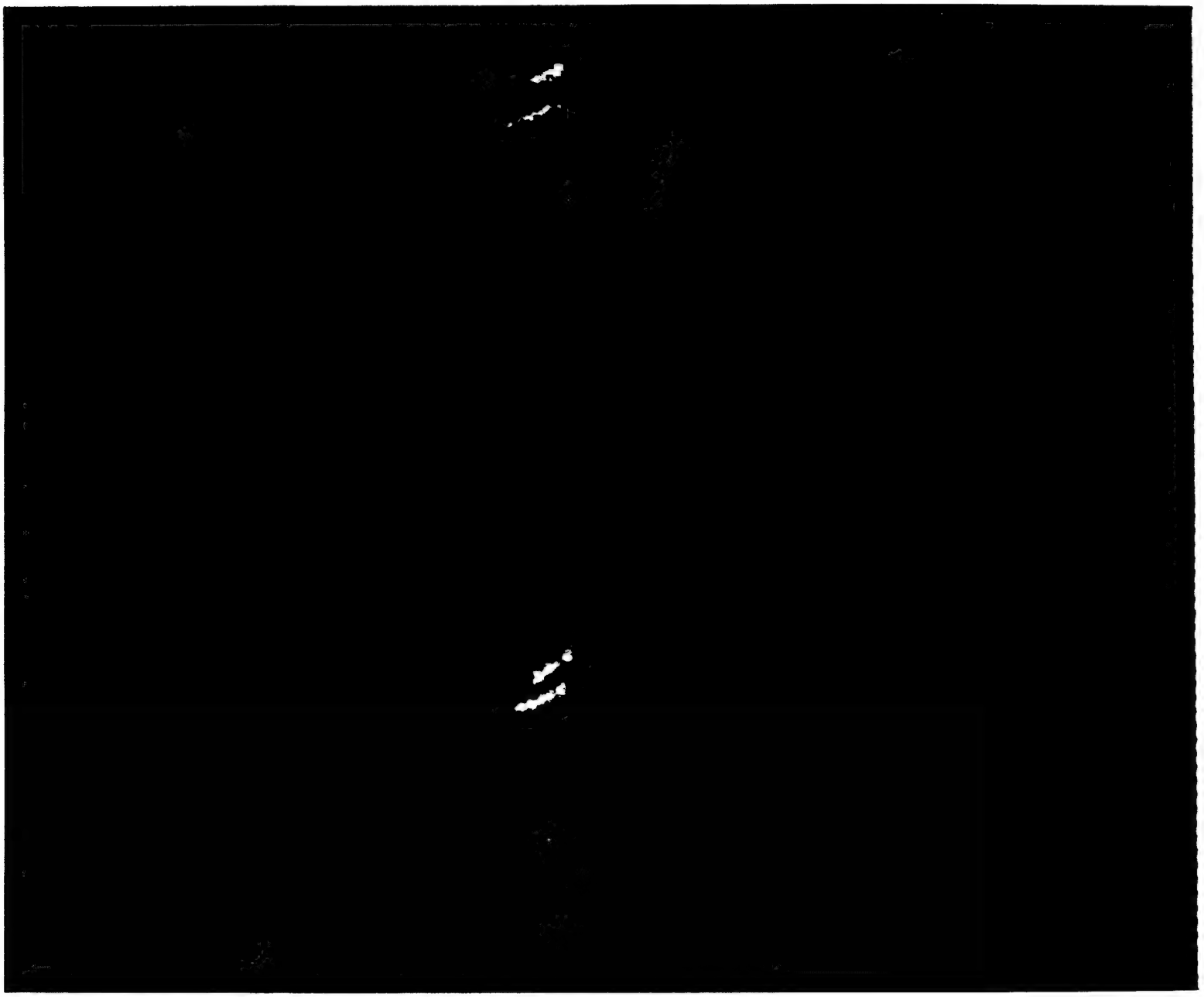


Figure 8.2: Viscous solution of one of the separating boosters from the DeltaII

Figure 8.3: Overall trajectory of booster tumbling from a fixed viewpoint

Figure 8.4: Density contour on cut through booster with rigid pole view at time t=0s.

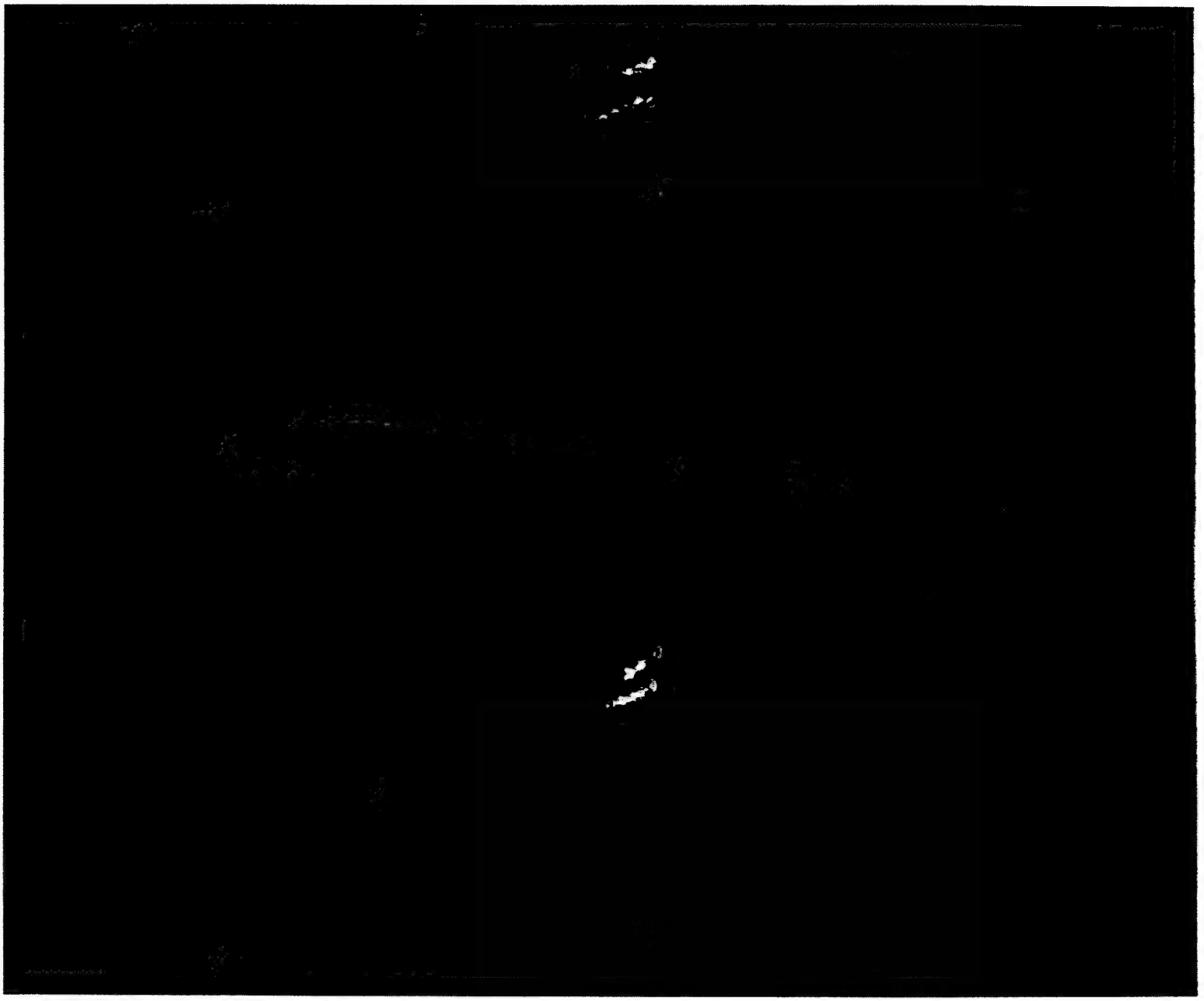


Figure 8.5: Density contour on cut through booster with rigid pole view at time t=21.8s.

Figure 8.6: Density contour on cut through booster with rigid pole view at time t=29.5s.

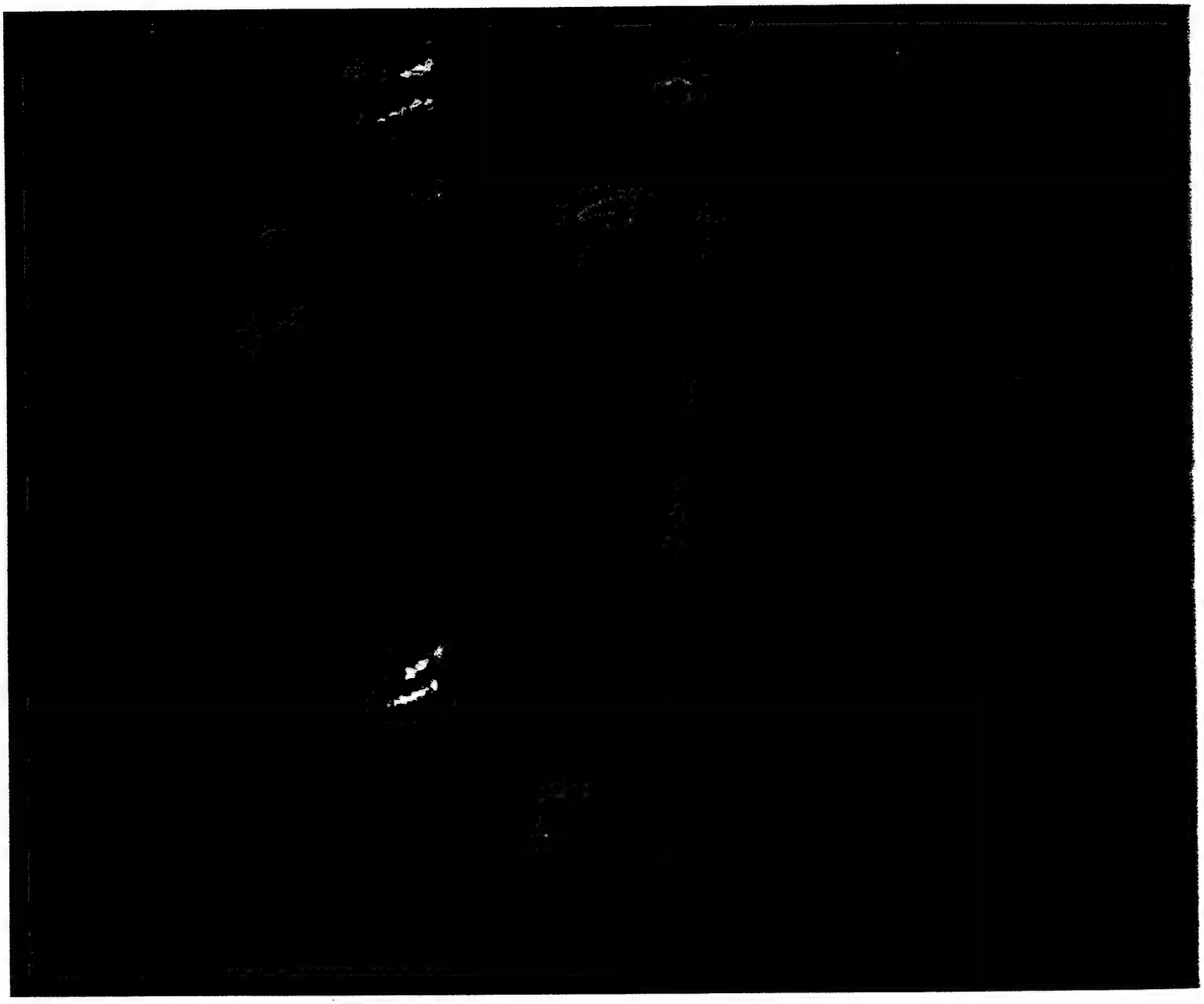


Figure 8.7: Contour on cut through booster with dynamic x,y static z view at time t=0.0s

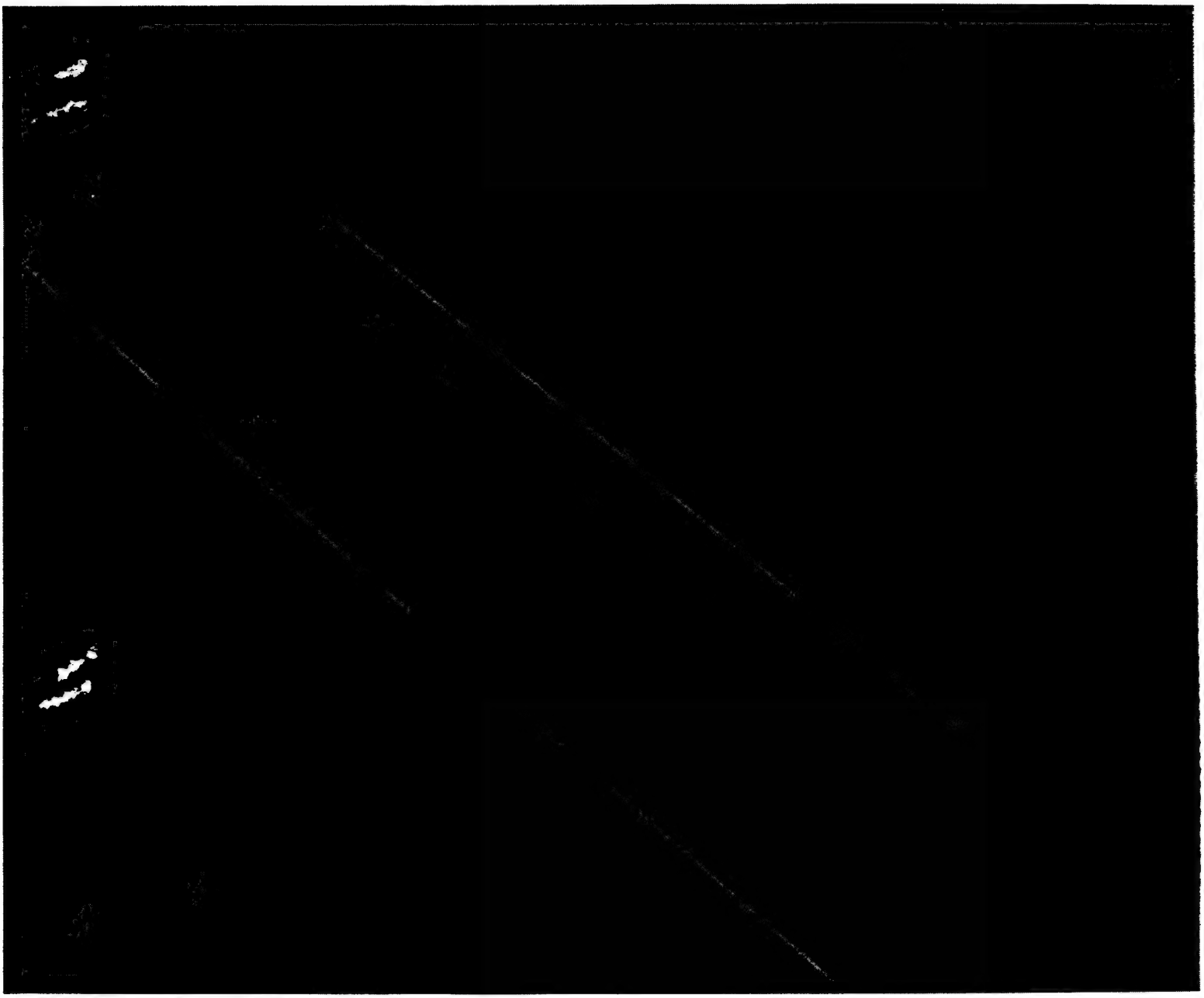


Figure 8.8: Contour on cut through booster with dynamic x,y static z view at time t=21.8s

Figure 8.9: Contour on cut through booster with dynamic x,y static z view at time t=29.5s

Figure 8.10: Contour on cuts through booster with dynamic x,y static z view at time t=0.0s

Figure 8.11: Contour on cuts through booster with dynamic x,y static z view at time t=21.8s

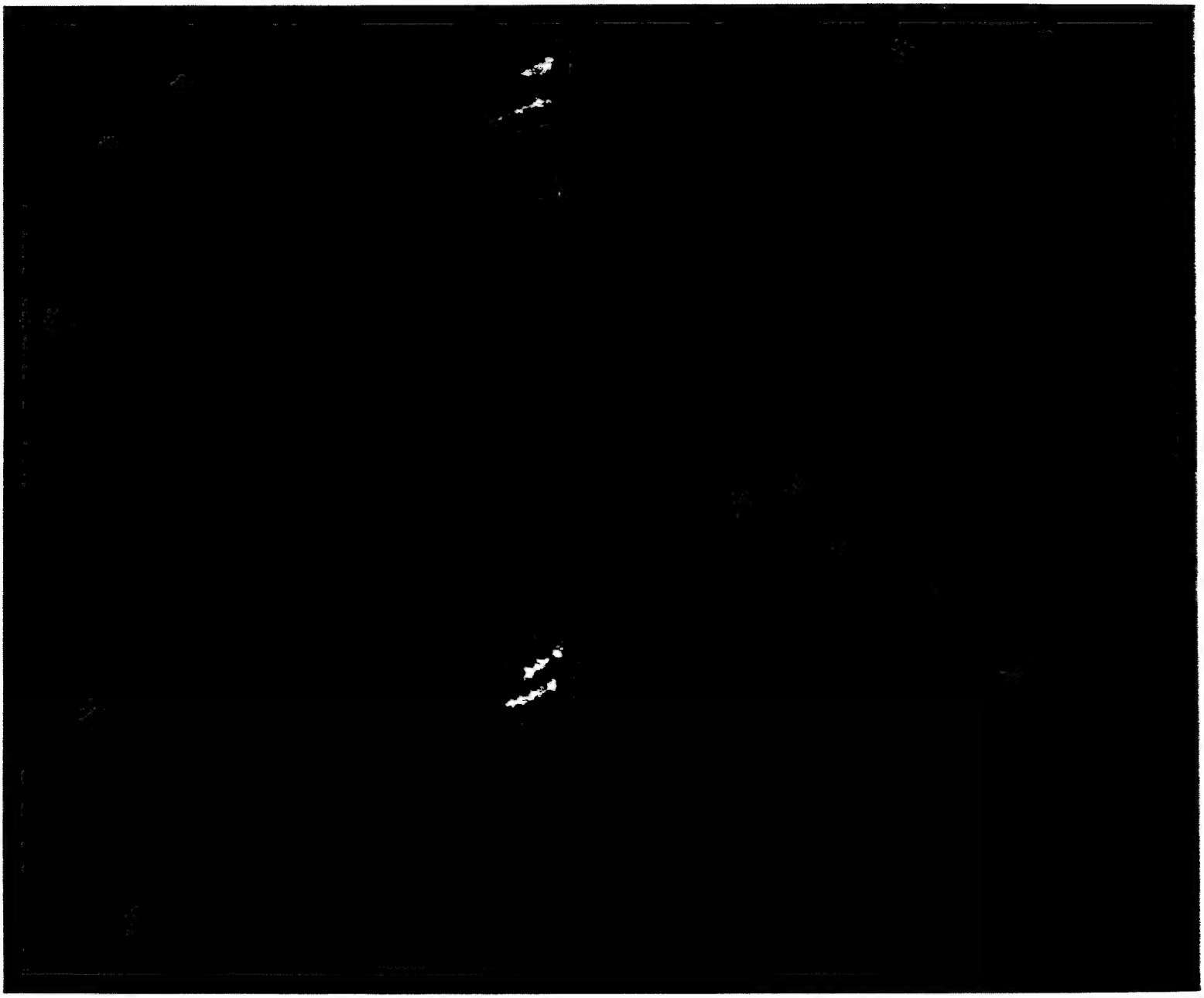


Figure 8.12: Contour on cuts through booster with dynamic x,y static z view at time t=29.5s

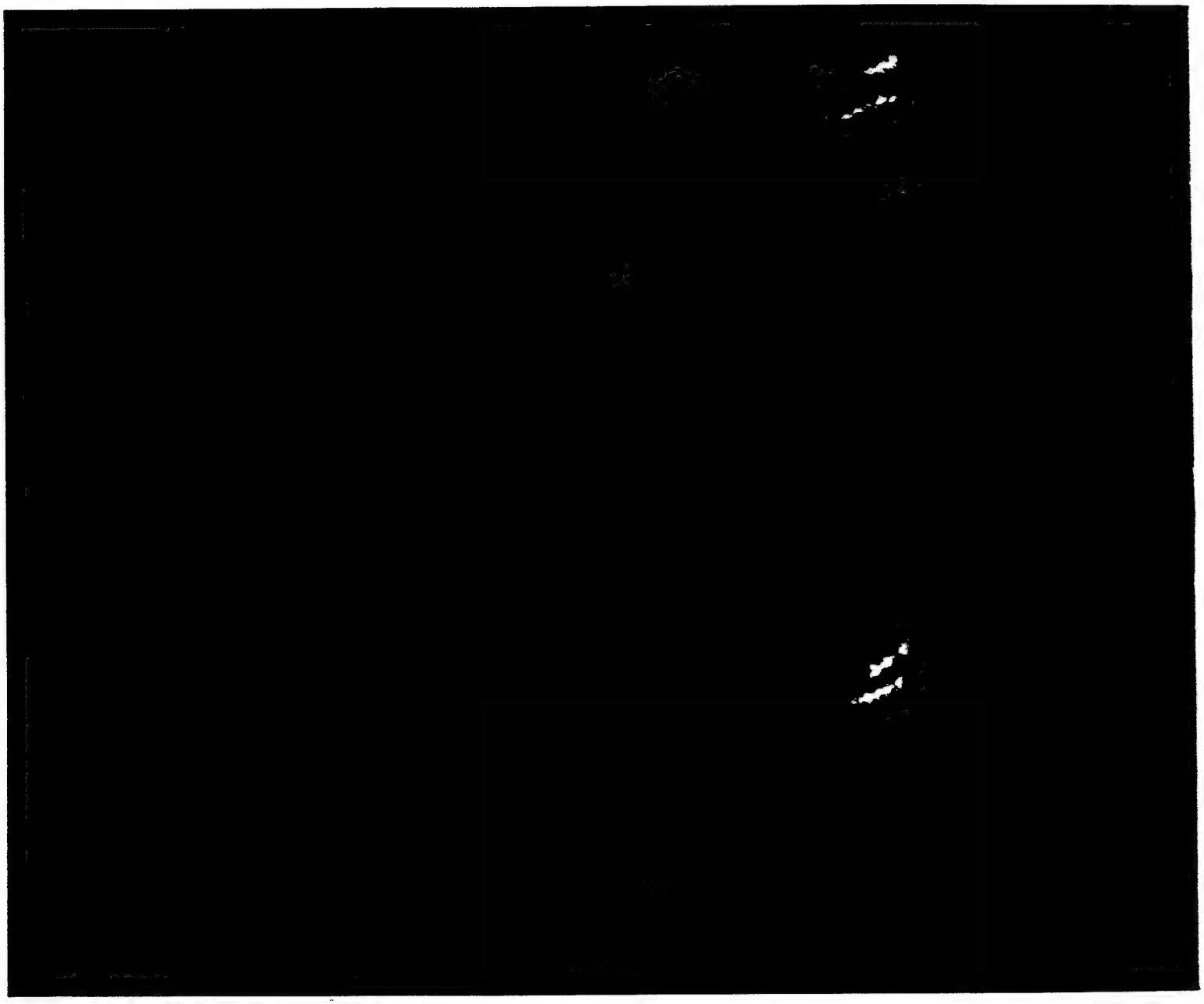


Figure 8.13: Isosurfaces of density on booster with dynamic x,y static z view at time t=0.0s

Figure 8.14: Isosurfaces of density on booster with dynamic x,y static z view at time t=21.8s

Figure 8.15: Isosurfaces of density on booster with dynamic x,y static z view at time t=29.5s

REFERENCES

[1] R. Briley. Computational fluid dynamics laboratory 2000 calendar. Technical report, Mississippi State University Engineering Research Center, 2000.

[2] M. Remotigue. *Structured Grid Technology to Enable Flow Simulation in an Integrated System*. PhD thesis, Mississippi State University, 1999.

[3] D. Marcum and A. Gaither. Unstructured grid generation for aerospace applications. Technical report, Proceedings of the ICASE/LaRC/ARO/NSF Workshop on Computational Aerosciences in the 21st Century, 1998.

[4] Jr. G. Newcomb. Graphical user interface designer: A guide for research and education. Master's thesis, Mississippi State University, 1997.

[5] O. Hassan, K. Marchant, E. Probert, N. P. Weatherill, and D. L. Marcum. The numerical simulation of 3d turbulent transonic flows using unstructured grids. Technical report, AIAA, 1994. AIAA-94-2346.

[6] R. Lohner. Matching semi-structured and unstructured grids for navier-stokes calculations. Technical report, AIAA, 1993. AIAA-93-3348.

[7] D. L. Marcum. Generation of unstructured grids for viscous flow applications. Technical report, AIAA, 1995. AIAA-95-0212.

[8] S. Pirzadeh. Unstructured viscous grid generation by the advancing-layers method. *AIAA Journal*, 32(8):1735, 1994.

[9] Y. Kallinderis, A. Khawaja, and H. McMorris. Hybrid prismatic/tetrahedral grid generation for viscous flows around complex geometries. *AIAA Journal*, 34(2):291, 1996.

[10] D. Sharov and K. Nakahasi. Hybrid prismatic/tetrahedral grid generation for viscous flow applications. *AIAA Journal*, 36(2):157, 1998.

[11] D. L. Marcum and N. P. Weatherill. Unstructured grid generation using iterative point insertion and local reconnection. *AIAA Journal*, 33(9):1619, 1995.

[12] D. L. Marcum. *The Handbook of Grid Generation*, chapter Unstructured grid generation using automatic point insertion and local reconnection, pages 18–1. CRC Press, 1999.

[13] J. A. Gaither. A solid modelling topology data structure for general grid generationn. Master's thesis, Mississippi State University, 1997.

[14] C. Sheng, D. Hyams, K. Sreenivas, A. Gaither, D. Marcum, and D. Whitfield. Three-dimensional incompressible navier-stokes flow computations about complete configurations using a multi-block unstructured grid approach. Technical report, AIAA, 1999. AIAA-99-0778.

[15] J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Comp. Phys.*, 72:449–266, 1987.

[16] Löhner. R. Adaptive remeshing for transient problems with moving bodies. Technical report, AIAA, 1988. AIAA-88-3737.

[17] J. D. Baum, Hong Luo, and R. Löhner. A new ale adaptive unstructured methodology for the simulation of moving bodies. Technical report, AIAA, 1994. AIAA-94-0414.

[18] K. P. Singh, J.C. Newman III, and O. Baysal. Dynamic unstructured method for flows past multiple bodies in relative motion. *AIAA Journal*, 33(4):641–646, 1995.

[19] J. T. Batina. Unsteady euler airfoil solutions using unstructured dynamic meshes. *AIAA Journal*, 28(8):1381–1388, 1990.

[20] R. Löhner. An adaptive finite element solver for transient problems with moving bodies. *Comp. Struct.*, 30:303–317, 1988.

[21] D. L. Marcum and N. P. Weatherill. Unstructured grid generation using iterative point insertion and local reconnection. *AIAA Journal*, 33(9):1619, 1995.

[22] D. L. Marcum and J. A. Gaither. Mixed element type unstructured grid generation for viscous flow applications. In *AIAA Computational Fluid Dynamics Conference*, 1999. AIAA-99-3252.

[23] Brian L. Stevens and Frank L. Lewis. *Aircraft control and simulation*. John Wiley & Sons Inc., 1992.

[24] Montgomery Hughson. *A 3-D unstructured CFD method for maneuvering vehicles*. PhD thesis, Mississippi State University, 1998.

[25] E. Howard Smart. *Advanced dynamics, Volume II, Dynamics of a solid body*. MacMillan & Co., limited, 1951.

[26] Andreas Haselbacher, James J. McGuirk, and Gary J. Page. Finite volume discretization aspects for viscous flows on mixed unstructured grids. *AIAA Journal*, 37(2):177–184, October 1999.

[27] David L. Marcum. Private conversations, Engineering Research Center for Computational Field Simulation, Mississippi State, MS, December 1997.

[28] D. L. Marcum. Unstructured grid technology me8993, mississippi state university, 1997.

[29] R. M. Beam and R. F. Warming. An implicit factored scheme for the compressible Navier-Stokes equations. *AIAA Journal*, 16(4):393–402, 1978.

[30] L. K. Taylor, J. A. Busby, A. Jiang, A. Arabshani, K. Sreenivas, and D. L. Whitfield. Time accurate incompressible Navier-Stokes simulation of the flapping foil experiment. In *Sixth International Conference on Numerical Ship Hydrodynamics*, 1993. Iowa City, Iowa, August.

[31] P. D. Thomas and C. K. Lombard. Geometric conservation law and its application to flow computations on moving grids. *AIAA Journal*, 17(10):1030–1037, 1978.

[32] M. J. Janus. *Advanced 3-D CFD Algorithm for Turbomachinery.* PhD thesis, Mississippi State University, 1989.

[33] P. R. Spalart and S. R. Allmaras. A one-equation turbulence model for aerodynamic flows. Technical report, AIAA, 1992. AIAA-92-0439.

[34] P. R. Spalart and M. Shur. On the sensitization of turbulence models to rotation and curvature. private publication, 1998.

[35] R. Pankajakshan. *Parallel Solution of Unsteady Incompressible Viscous Flows Using Multiblock Structured Grids.* PhD thesis, Mississippi State University, 1997.

[36] C. Sheng, D. L. Whitfield, and W. K. Anderson. Multiblock approach for calculating incompressible fluid flows on unstructured grids. *AIAA Journal*, 37(2):169–176, 1999.

[37] M. K. Bhat. Parallel implementation of an unstructured grid-based Navier-Stokes solver. In *37th Aerospace Sciences Meeting and Exhibit*, 1999. AIAA Paper 99-0663, Reno, NV., Jan.

[38] G. Karypis and V. Kumar. METIS: Family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis/metis.html, 1998.

[39] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical report, University of Minnesota, 1998. TR 98-019, Department of Computer Science.

[40] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[41] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM journal on Scientific Computing*, 1998.

[42] R. R. Springmeyer, M. M. Blattner, and N. L. Max. A characterization of the scientific data analysis process. *Proceedings of IEEE Visualization 1992*, pages 235–242, October 1992.

[43] W. J. Schroeder. Adapting data-flow systems to large datasets. *Published in ACM SIGGRAPH Course Notes: System Designs for Visualizing Large Scale Scientific Data*, pages 31–46, August 1999.

[44] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. *Proceedings of IEEE Visualization '97*, October 1997.

[45] M. Cox and D. Ellsworth. Managing big data for scientific visualization. *Published in ACM SIGGRAPH Course Notes: Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design*, August 1997.

[46] J. Bloomer. *Power Programming with RPC.* O'Reilly & Associates, Inc., 1992.

[47] S. Bryson, D. Kenwright, and M. Gerald-Yamasaki. Fel: The field encapsulation library. *Proceedings of IEEE Visualization '96*, pages 241–247, October 1996.

[48] G. V. Bancroft, F. J. Merritt, T. C. Plessell, P. G. Kelaita, R. K. McCabe, and A. Globus. Fast: A multi-processed environment for visualization. *Proceedings of IEEE Visualization '90*, pages 14–27, October 1990.

[49] Personal Communications with Dr. David Marcum, Mississippi State University.

[50] D. Marcum. *ME8993 Unstructured Grid Generation Technical Class Notes*. Mississippi State University, 1995. Fall Semester.

[51] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics*, 4(1):41–59, January 1985.

[52] K.-Y. Whang, J.-W. Song, J.-W. Chang, J.-Y. Kim, W.-S. Cho, C.-M. Park, and I.-Y. Song. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, December 1995.

[53] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-based decimation of marching cubes surfaces. *Proceedings of IEEE Visualization 1996*, pages 335–342, October 1996.

[54] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics*, 25(4):275–284, July 1991.

[55] Y. Livnat, H.-W. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, April 1996.

[56] R. S. Gallagher. Span filter: An optimization scheme for volume visualization of large finite element models. *Proceedings of IEEE Visualization 1991*, pages 68–75, October 1991.

[57] P. Cignoni, P. Marino, and C. Montani. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April 1997.

[58] M. Giles and R. Haimes. Advanced interactive visualization for cfd. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[59] H. Shen and C. Johnson. Sweeping simplices: A fast isosurface extraction algorithm for unstructured grids. *Proceedings of IEEE Visualization 1995*, pages 143–150, October 1995.

[60] H. Shen, C. Hansen, Y. Livnat, and C. Johnson. Isosurfacing in span space with utmost efficiency (issue). *Proceedings of IEEE Visualization 1996*, pages 287–294, October 1996.

[61] R. Lohner, P. Parick, and C. Gumbert. Some algorithmic problems of plotting codes in unstructured grids. Technical report, AIAA, 1989. AIAA 89-1981.

[62] Chris Young and Drew Wells. *Ray Tracing Creations*. Waite Group Press, 2nd edition, 1994.

APPENDIX A

GRID INPUT FORMATS

The DIVA visualization system is capable of handling structured multi-block, unstructured tetrahedral, and unstructured mixed element grids. The file readers associated with the DIVA visualization system key off of a suffix appended to the end of the grid file name. A structured multi-block grid has the suffix .sgrid, an unstructured tetrahedral grid has the suffix .fgrid, and an unstructured mixed element grid has the suffix .ugrid.

Structured Multi-block

The format for a structured multi-block grid is as follows:

NumBlocks

for i = 1 to NumBlocks, read in NumI, NumJ, NumK Dimensions for Block i

for i = 1 to NumBlocks, for j = 1 to NumI*NumJ*NumK, read X,Y,Z

Unstructured Tetrahedral

The format for an unstructured tetrahedral grid is as follows:

NumNodes, NumSurfTriangles, NumVolTets

for i = 1 to NumNodes, read in all X Coordinates

for i = 1 to NumNodes, read in all Y Coordinates

for i = 1 to NumNodes, read in all Z Coordinates

for i = 1 to NumSurfTriangles, read in three indices into coordinates

for i = 1 to NumSurfTriangles, read in boundary condition for triangle i

for i = 1 to NumVolTets, read in four indices into coordinates

Unstructured Mixed Element

The format for an unstructured mixed element grid is as follows:

NumNodes, NumSurfTriangles, NumSurfQuads,

NumVolTets, NumVolPents5, NumVolPents6, NumVolHexs

for i = 1 to NumNodes, read in all X Coordinates

for i = 1 to NumNodes, read in all Y Coordinates

for i = 1 to NumNodes, read in all Z Coordinates

for i = 1 to NumSurfTriangles, read in three indices into coordinates

for i = 1 to NumSurfQuads, read in four indices into coordinates

for i = 1 to NumSurfTriangles+NumSurfQuads, read in boundary condition for surface i

for i = 1 to NumVolTets, read in four indices into coordinates

for i = 1 to NumVolPents5, read in five indices into coordinates

for i = 1 to NumVolPents6, read in six indices into coordinates

for i = 1 to NumVolHexs, read in eight indices into coordinates

APPENDIX B

SOLUTION INPUT FORMATS

The DIVA visualization system is capable of handling the standard Q file, the incompressible Q file, and the function file formats for solution input. The file readers associated with the DIVA visualization system key off of a suffix appended to the end of the solution file name. A structured multi-block standard Q file has the suffix .sflow, an unstructured tetrahedral standard Q file has the suffix .fflow, and an unstructured mixed element standard Q file has the suffix .uflow. A structured multi-block incompressible Q file has the suffix .zflow. A structured multi-block function file has the suffix .sfunc, an unstructured tetrahedral function file has the suffix .ffunc or .unfunc.

Structured Multi-block Standard Q File

The format for a structured multi-block standard Q file is as follows:

NumBlocks

for i = 1 to NumBlocks, read in NumI, NumJ, NumK Dimensions for Block i

for i = 1 to NumBlocks,

read four doubles RefMach, Alpha, RefReynolds, Time

for j = 1 to NumI*NumJ*NumK,

read all block i's density values

read all block i's density*u velocity values

read all block i's density*v velocity values

read all block i's density*w velocity values

read all block i's energy values

Unstructured Standard Q File

The format for a unstructured standard Q file is as follows:

NumNodes, NumJNodes, NumKNodes

read four doubles RefMach, Alpha, RefReynolds, Time

for i = 1 to NumNodes,

read all node i's density values

read all node i's density*u velocity values

read all node i's density*v velocity values

read all node i's density*w velocity values

read all node i's energy values

Structured Multi-block Incompressible Q File

The format for a structured multi-block incompressible Q file is as follows:

NumBlocks

for i = 1 to NumBlocks, read in NumI, NumJ, NumK Dimensions for Block i

for i = 1 to NumBlocks,

read four doubles RefMach, Alpha, RefReynolds, Time

for j = 1 to NumI*NumJ*NumK,

read all block i's Q1 values

read all block i's u velocity values

read all block i's v velocity values

read all block i's w velocity values

read all block i's Q5 values

Structured Multi-block Function File

The format for a structured multi-block function file is as follows:

NumFunctions,NumBlocks

for i = 1 to NumFunctions, read in whether function is scalar 1 or vector 0

for i = 1 to NumBlocks, read in NumI, NumJ, NumK Dimensions for Block i

for i = 1 to NumFunctions,

for j = 1 to NumBlocks,

for k = 1 to NumI*NumJ*NumK for block j,

if the function is a vector, read three values

if the function is a scalar, read one value

Unstructured Function File

The format for an unstructured function file is as follows:

NumFunctions,

for i = 1 to NumFunctions,

read NumFunctionNodes, NumJNodes, NumKNodes, FunctionFlag for function i

for i = 1 to NumBlocks, read in NumI, NumJ, NumK Dimensions for Block i

for i = 1 to NumFunctions,

for j = 1 to NumFunctionNodes for function i

if the function is a vector, read three values

if the function is a scalar, read one value

APPENDIX C

ALGORITHM TO FIND ELEMENTS SURROUNDING A POINT

```
void elementsSurroundingPoint( void )
{
    int     i       = 0;    // i is a counting variable.
    int     j       = 0;    // j is a counting variable.
    int     pIndex  = 0;    // pIndex is a variable used to store a point index.
    int     eIndex  = 0;    // eIndex is a variable used to store an element index.
    INT_1D  *tNESP  = NULL; // tNESP is a temporary array for constructing nESP.
    INT_1D  *nESP   = NULL; // nESP contains information for number of elements
                            // surrounding a point.
    INT_1D  *eSP    = NULL; // eSP contains the specific elements surrounding each points.

    nESP = new INT_1D[numberOfNodes+1];  // Allocate memory for actual array.
    tNESP = new INT_1D[numberOfNodes+1]; // Allocate memory for temporary array.
    for (i = 0; i <= numberOfNodes; i++)      // For i cycles over all nodes in grid plus one.
    {
        tNESP[i] = 0;                         // Initialize temporary array to contain zeros.
        nESP[i] = 0;                          // Initialize actual array to contain zeros.
    }                                         // End cycle over all nodes in grid plus one.

    for (i = 0; i < numTetrahedra; i++)  // For i cycles over all tetrahedra in the grid.
    {
        for (j = 0; j < 4; j++)           // For j cycles over all nodes in tetrahedra i.
        {
            pIndex = tetrahedra[i][j];    // Dereference the node index.
            tNESP[pIndex]++;              // Increment this nodes number of elements by one.
        }                                 // End cycle j over all nodes in tetrahedra i.
    }                                     // End cycle i over all tetrahedra in the grid.

    for (i = 0; i < numPents5; i++)  // For i cycles over all five noded pents in the grid.
    {
        for (j = 0; j < 5; j++)           // For j cycles over all nodes in five noded pent i.
        {
            pIndex = pents5[i][j];    // Dereference the node index.
            tNESP[pIndex]++;          // Increment this nodes number of elements by one.
        }                             // End cycle j over all nodes in five noded pent i.
    }                                 // End cycle i over all five noded pents in the grid.

    for (i = 0; i < numPents6; i++)  // For i cycles over all six noded pents in the grid.
    {
        for (j = 0; j < 6; j++)           // For j cycles over all nodes in six noded pent i.
        {
            pIndex = pents6[i][j];    // Dereference the node index.
            tNESP[pIndex]++;          // Increment this nodes number of elements by one.
        }                             // End cycle j over all nodes in six noded pent i.
    }                                 // End cycle over all six noded pents in the grid.
```

```
for (i = 0; i < numHexahedra; i++)  // For i cycles over all hexahedra in the grid.
{
    for (j = 0; j < 8; j++)         // For j cycles over all nodes in the hexahedra i.
    {
        pIndex = hexahedra[i][j];   // Dereference the node index.
        tNESP[pIndex]++;            // Increment this nodes number of elements by one.
    }                               // End cycle j over all nodes in hexahedra i.
}                                   // End cycle over all hexahedra in the grid.

for (i = 1; i <= numberOfNodes; i++)  // For i cycles over all nodes plus one starting
                                      // at one.
    nESP[i] = nESP[i-1] + tNESP[i-1]; // Increment nESP[i] to indicate all elements
                                      // surrounding a point up to that point.
delete[] tNESP;                       // Deallocate the memory needed for this
                                      // temporary variable.

eSP = new INT_1D[nESP[numberOfNodes]]; // Allocate the memory for actual element
                                       // indices.

for (i = 0; i < numTetrahedra; i++)   // For i cycles over all tetrahedra in the grid.
{
    for (j = 0; j < 4; j++)           // For j cycles over all nodes in tetrahedra i.
    {
        pIndex = tetrahedra[i][j];    // Dereference the node index.
        eIndex = nESP[pIndex];        // Dereference the current element index.
        eSP[eIndex] = tetIndex+i;     // Add this element to the list surrounding node
                                      // pIndex.
        nESP[pIndex]++;               // Increment this nodes number of elements by one.
    }                                 // End cycle j over all nodes in tetrahedra i.
}                                     // End cycle i over all tetrahedra in the grid.

for (i = 0; i < numPents5; i++)       // For i cycles over all five noded pents in the grid.
{
    for (j = 0; j < 5; j++)           // For j cycles over all nodes in five noded pent i.
    {
        pIndex = pents5[i][j];        // Dereference the node index.
        eIndex = nESP[pIndex];        // Dereference the current element index.
        eSP[eIndex] = pents5Index+i;  // Add this element to the list surrounding node
                                      // pIndex.
        nESP[pIndex]++;               // Increment this nodes number of elements by one.
    }                                 // End cycle j over all nodes in five noded pents i.
}                                     // End cycle i over all five noded pents in the grid.
```

```
    for (i = 0; i < numPents6; i++)          // For i cycles over all six noded pents in the grid.
    {
        for (j = 0; j < 6; j++)              // For j cycles over all nodes in six noded pent i.
        {
            pIndex = pents6[i][j];           // Dereference the node index.
            eIndex = nESP[pIndex];           // Dereference the current element index.
            eSP[eIndex] = pents6Index+i;     // Add this element to the list surrounding node
                                             // pIndex.
            nESP[pIndex]++;                  // Increment this nodes number of elements by one.
        }                                    // End cycle j over all nodes in six noded pents i.
    }                                        // End cycle i over all six noded pents in the grid.

    for (i = 0; i < numHexahedra; i++)       // For i cycles over all hexahedra in the grid.
    {
        for (j = 0; j < 8; j++)              // For j cycles over all nodes in hexahedra i.
        {
            pIndex = hexahedra[i][j];        // Dereference the node index.
            eIndex = nESP[pIndex];           // Dereference the current element index.
            eSP[eIndex] = hexIndex+i;        // Add this element to the list surrounding node
                                             // pIndex.
            nESP[pIndex]++;                  // Increment this nodes number of elements by one.
        }                                    // End cycle j over all nodes in hexahedra i.
    }                                        // End cycle i over all hexahedra in the grid.
}                                            // End compute elements surrounding point.
```

APPENDIX D

ALGORITHM TO FIND ELEMENT NEIGHBORS

```
void buildElementNeighborMap( void )
{
    int     i   = 0;      // i is a counting variable.
    int     j   = 0;      // j is a counting variable.
    int     p1  = 0;      // p1 is used as a node index.
    int     p2  = 0;      // p2 is used as a node index.
    int     p3  = 0;      // p3 is used as a node index.
    int     p4  = 0;      // p4 is used as a node index.
    int     p5  = 0;      // p5 is used as a node index.
    int     p6  = 0;      // p6 is used as a node index.
    int     ne  = 0;      // ne is used as a holder for a matching element.
    int     nf  = 0;      // nf is used as a holder for a matching face.
    int     ge  = 0;      // ge is used to hold the global element index.
    INT_6D  *eN= NULL;  // eN is the array containing explicit element neighbor information.

    eN = new INT_6D[numElements];  // Allocate memory for the element neighbor array.

    for (i = 0; i < numElements; i++)   // For i cycles over all elements in the grid.
    {
        eN[i][0] = -555; // Initialize the neighbor in position 0 to value indicating not visited.
        eN[i][1] = -555; // Initialize the neighbor in position 1 to value indicating not visited.
        eN[i][2] = -555; // Initialize the neighbor in position 2 to value indicating not visited.
        eN[i][3] = -555; // Initialize the neighbor in position 3 to value indicating not visited.
        eN[i][4] = -555; // Initialize the neighbor in position 4 to value indicating not visited.
        eN[i][5] = -555; // Initialize the neighbor in position 5 to value indicating not visited.
    }

    for (i = 0; i < numTetrahedra; i++)   // For i cycles over all tetrahedra in the grid.
    {
        p1 = tetrahedra[i][0];   // p1 contains an index to node 1 of tetrahedra i.
        p2 = tetrahedra[i][1];   // p2 contains an index to node 2 of tetrahedra i.
        p3 = tetrahedra[i][2];   // p3 contains an index to node 3 of tetrahedra i.
        p4 = tetrahedra[i][3];   // p4 contains an index to node 4 of tetrahedra i.
        ge = tetIndex+i;         // ge contains the global element index.

        if (eN[ge][0] == -555)    // If neighbor 1 for tetrahedra i has not been visited,
        {
            // Find the neighbor element and face for neighbor 1 of the tetrahedra.
            commonElement(ge,p1,p2,p3,&ne,&nf);
            eN[ge][0] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 1 for tetrahedra i has not been visited.
```

```
    if (eN[ge][1] == -555)    // If neighbor 2 for tetrahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 2 of the tetrahedra.
        commonElement(ge,p2,p3,p4,&ne,&nf);
        eN[ge][1] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 2 for tetrahedra i has not been visited.

    if (eN[ge][2] == -555)    // If neighbor 3 for tetrahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 3 of the tetrahedra.
        commonElement(ge,p3,p4,p1,&ne,&nf);
        eN[ge][2] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 3 for tetrahedra i has not been visited.
    if (eN[ge][3] == -555)    // If neighbor 4 for tetrahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 4 of the tetrahedra.
        commonElement(ge,p4,p2,p1,&ne,&nf);
        eN[ge][3] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 4 for tetrahedra i has not been visited.
}                             // End for i cycles over all tetrahedra in the grid.
for (i = 0; i < numPents5; i++)       // For i cycles over all five noded pents in the grid.
{
    p1 = pents5[i][0];        // p1 contains an index to node 1 of five noded pent i.
    p2 = pents5[i][1];        // p2 contains an index to node 2 of five noded pent i.
    p3 = pents5[i][2];        // p3 contains an index to node 3 of five noded pent i.
    p4 = pents5[i][3];        // p4 contains an index to node 4 of five noded pent i.
    p5 = pents5[i][4];        // p5 contains an index to node 5 of five noded pent i.
    ge = pents5Index+i;       // ge contains the global element index.


    if (eN[ge][0] == -555) // If neighbor 1 for five noded pent i has not been visited,
    {
        // Find the neighbor element and face for neighbor 1 of the five noded pent.
        commonElement(ge,p1,p2,p3,&ne,&nf);
        eN[ge][0] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 1 for five noded pent i has not been visited.
```

```
        if (eN[ge][1] == -555)    // If neighbor 2 for five noded pent i has not been visited,
        {
            // Find the neighbor element and face for neighbor 2 of the five noded pent.
            commonElement(ge,p2,p5,p3,&ne,&nf);
            eN[ge][1] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 2 for five noded pent i has not been visited.
        if (eN[ge][2] == -555)    // If neighbor 3 for five noded pent i has not been visited.
        {
            // Find the neighbor element and face for neighbor 3 of the five noded pent.
            commonElement(ge,p5,p4,p3,&ne,&nf);
            eN[ge][2] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 3 for five noded pent i has not been visited.

        if (eN[ge][3] == -555)    // If neighbor 4 for five noded pent i has not been visited.
        {
            // Find the neighbor element and face for neighbor 4 of the five noded pent.
            commonElement(ge,p3,p4,p1,&ne,&nf);
            eN[ge][3] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 3 for five noded pent i has not been visited.
        if (eN[ge][4] == -555)    // If neighbor 5 for five noded pent i has not been visited,
        {
            // Find the neighbor element and face for neighbor 5 of the five noded pent.
            commonElement(ge,p1,p4,p5,p2,&ne,&nf);
            eN[ge][4] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 5 for five noded pent i has not been visited.
    }                             // End for i cycles over all five noded pent in the grid.
    for (i = 0; i < numPents6; i++)       // For i cycles over all six noded pents in the grid.
    {
        p1 = pents6[i][0];        // p1 contains an index to node 1 of six noded pent i.
        p2 = pents6[i][1];        // p2 contains an index to node 2 of six noded pent i.
        p3 = pents6[i][2];        // p3 contains an index to node 3 of six noded pent i.
        p4 = pents6[i][3];        // p4 contains an index to node 4 of six noded pent i.
        p5 = pents6[i][4];        // p5 contains an index to node 5 of six noded pent i.
        p6 = pents6[i][5];        // p6 contains an index to node 6 of six noded pent i.
        ge = pents6Index+i;       // ge contains the global element index.
```

```
if (eN[ge][0] == -555)    // If neighbor 1 for six noded pent i has not been visited,
{
    // Find the neighbor element and face for neighbor 1 of the six noded pent.
    commonElement(ge,p1,p2,p3,&ne,&nf);
    eN[ge][0] = ne;       // Record the information in the element neighbor array.
    if (ne != -999)       // If the neighbor element indicates an actual face,
        eN[ne][nf] = ge; // Save the information for the neighbor also.
}                         // End if neighbor 1 for six noded pent i has not been visited.
if (eN[ge][1] == -555)    // If neighbor 2 for six noded pent i has not been visited,
{
    // Find the neighbor element and face for neighbor 2 of the six noded pent.
    commonElement(ge,p2,p3,p6,p5,&ne,&nf);
    eN[ge][1] = ne;       // Record the information in the element neighbor array.
    if (ne != -999)       // If the neighbor element indicates an actual face,
        eN[ne][nf] = ge; // Save the information for the neighbor also.
}                         // End if neighbor 2 for six noded pent i has not been visited.
if (eN[ge][2] == -555)    // If neighbor 3 for six noded pent i has not been visited,
{
    // Find the neighbor element and face for neighbor 3 of the six noded pent.
    commonElement(ge,p4,p6,p5,&ne,&nf);
    eN[ge][2] = ne;       // Record the information in the element neighbor array.
    if (ne != -999)       // If the neighbor element indicates an actual face,
        eN[ne][nf] = ge; // Save the information for the neighbor also.
}                         // End if neighbor 3 for six noded pent i has not been visited.


if (eN[ge][3] == -555)    // If neighbor 4 for six noded pent i has not been visited,
{
    // Find the neighbor element and face for neighbor 4 of the six noded pent.
    commonElement(ge,p4,p1,p3,p6,&ne,&nf);
    eN[ge][3] = ne;       // Record the information in the element neighbor array.
    if (ne != -999)       // If the neighbor element indicates an actual face,
        eN[ne][nf] = ge; // Save the information for the neighbor also.
}                         // End if neighbor 3 for six noded pent i has not been visited.
if (eN[ge][4] == -555)    // If neighbor 5 for six noded pent i has not been visited,
{
    // Find the neighbor element and face for neighbor 5 of the six noded pent.
    commonElement(ge,p2,p1,p4,p5,&ne,&nf);
    eN[ge][4] = ne;       // Record the information in the element neighbor array.
    if (ne != -999)       // If the neighbor element indicates an actual face,
        eN[ne][nf] = ge; // Save the information for the neighbor also.
}                         // End if neighbor 5 for six noded pent i has not been visited.
}                         // End for i cycles over all six noded pent in the grid.
```

```
for (i = 0; i < numHexahedra; i++)  // For i cycles over all hexahedra in the grid.
{
    p1 = hexahedra[i][0];     // p1 contains an index to node 1 of hexahedra i.
    p2 = hexahedra[i][1];     // p2 contains an index to node 2 of hexahedra i.
    p3 = hexahedra[i][2];     // p3 contains an index to node 3 of hexahedra i.
    p4 = hexahedra[i][3];     // p4 contains an index to node 4 of hexahedra i.
    p5 = hexahedra[i][4];     // p5 contains an index to node 5 of hexahedra i.
    p6 = hexahedra[i][5];     // p6 contains an index to node 6 of hexahedra i.
    p7 = hexahedra[i][6];     // p7 contains an index to node 7 of hexahedra i.
    p8 = hexahedra[i][7];     // p8 contains an index to node 8 of hexahedra i.
    ge = hexIndex+i;          // ge contains the global element index.

    if (eN[ge][0] == -555)    // If neighbor 1 for hexahedra i has not been visited.
    {
        // Find the neighbor element and face for neighbor 1 of the hexahedra.
        commonElement(ge,p1,p2,p3,p4,&ne,&nf);
        eN[ge][0] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 1 for hexahedra i has not been visited.
    if (eN[ge][1] == -555)    // If neighbor 2 for hexahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 2 of the hexahedra.
        commonElement(ge,p3,p7,p6,p2,&ne,&nf);
        eN[ge][1] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 2 for hexahedra i has not been visited.
    if (eN[ge][2] == -555)    // If neighbor 3 for hexahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 3 of the hexahedra.
        commonElement(ge,p6,p5,p8,p7,&ne,&nf);
        eN[ge][2] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 3 for hexahedra i has not been visited.

    if (eN[ge][3] == -555)    // If neighbor 4 for hexahedra i has not been visited,
    {
        // Find the neighbor element and face for neighbor 4 of the hexahedra.
        commonElement(ge,p1,p4,p8,p5,&ne,&nf);
        eN[ge][3] = ne;       // Record the information in the element neighbor array.
        if (ne != -999)       // If the neighbor element indicates an actual face,
            eN[ne][nf] = ge; // Save the information for the neighbor also.
    }                         // End if neighbor 3 for hexahedra i has not been visited.
```

```
        if (eN[ge][4] == -555)    // If neighbor 5 for hexahedra i has not been visited,
        {
            // Find the neighbor element and face for neighbor 5 of the hexahedra.
            commonElement(ge,p4,p3,p7,p8,&ne,&nf);
            eN[ge][4] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 5 for hexahedra i has not been visited.
        if (eN[ge][5] == -555)    // If neighbor 6 for hexahedra i has not been visited,
        {
            // Find the neighbor element and face for neighbor 6 of the hexahedra.
            commonElement(ge,p1,p5,p6,p2,&ne,&nf);
            eN[ge][5] = ne;       // Record the information in the element neighbor array.
            if (ne != -999)       // If the neighbor element indicates an actual face,
                eN[ne][nf] = ge; // Save the information for the neighbor also.
        }                         // End if neighbor 6 for hexahedra i has not been visited.
    }                             // End for i cycles over all hexahedra in the grid.
}                                 // End buildElementNeighborMap.


void commonElement( int insideThisElement, // The global id for the element we are inside.
                    int p1, // The first point index for the face we want to match.
                    int p2, // The second point index for the face we want to match.
                    int p3, // The third point index for the face we want to match.
                    int *neighborElement, // Returns the matching element id.
                    int *neighborElementFace ) // Returns the matching face id.
{
    if (nESP == NULL or eSP == NULL)    // Check to see if maps are created.
        createElementsSurroundingPoint(); // If not, then create it.
    *neighborElement = -999;            // Initialize return value to -999.
    *neighborElementFace = -999;        // Do the same with neighborElementFace.

    int   i          = 0; // i is used as a holder.
    int   n1         = 0; // n1 is the number of elements surrounding point p1.
    int   done       = 0; // done is a variable used as a stopping condition.
    int   sIndex     = 0; // sIndex is the starting index for p1 into eSP.
    int   positionP1 = 0; // positionP1 will contain the position on the matching face.
    int   positionP2 = 0; // positionP2 will contain the position on the matching face.
    int   positionP3 = 0; // positionP3 will contain the position on the matching face.
    if (p1 == 0)
    {
        n1 = nESP[p1]; // n1 is set to nESP[0] if p1 is 0.
        sIndex = 0;    // sIndex is also set to 0.
    }
```

```
else
{
    n1 = nESP[p1] - nESP[p1-1]; // n1 is set by finding difference in values in nESP.
    sIndex = nESP[p1-1];        // sIndex is the value at index p1-1.
}

i = sIndex;   // Let i cycle over all elements.
done = 0;     // Initialize stopping condition to 0.
while (!done and i < sIndex+n)   // Cycle until a match or no more elements to check.
{
    positionP1 = -1;        // Initialize positionP1.
    positionP2 = -1;        // Initialize positionP2.
    positionP3 = -1;        // Initialize positionP3.

    if (eSP[i] != insideThisElement) // Check for found a match.
    {
        if (eSP[i] >= tetIndex and eSP[i] < pents5Index) // Check for match in tetrahedra.
        {
            if (tetrahedra[eSP[i]][0] == p1)
                positionP1 = 0;
            else if (tetrahedra[eSP[i]][1] == p1)
                positionP1 = 1;
            else if (tetrahedra[eSP[i]][2] == p1)
                positionP1 = 2;
            else if (tetrahedra[eSP[i]][3] == p1)
                positionP1 = 3;

            if (tetrahedra[eSP[i]][0] == p2)
                positionP2 = 0;
            else if (tetrahedra[eSP[i]][1] == p2)
                positionP2 = 1;
            else if (tetrahedra[eSP[i]][2] == p2)
                positionP2 = 2;
            else if (tetrahedra[eSP[i]][3] == p2)
                positionP2 = 3;

            if (tetrahedra[eSP[i]][0] == p3)
                positionP3 = 0;
            else if (tetrahedra[eSP[i]][1] == p3)
                positionP3 = 1;
            else if (tetrahedra[eSP[i]][2] == p3)
                positionP3 = 2;
            else if (tetrahedra[eSP[i]][3] == p3)
```

```
            positionP3 = 3;

        if (positionP1 >= 0 and positionP2 >= 0 and positionP3 >= 0)
        {
            *neighborElement = eSP[i];
            if (positionP1 != 3 and positionP2 != 3 and positionP3 != 3)
                *neighborElementFace = 0;
            else if (positionP1 != 0 and positionP2 != 0 and positionP3 != 0)
                *neighborElementFace = 1;
            else if (positionP1 != 1 and positionP2 != 1 and positionP3 != 1)
                *neighborElementFace = 2;
            else
                *neighborElementFace = 3;

            done = 1; // Set stopping condition to 1.
        }
    } // End if (eSP[i] >= tetIndex and eSP[i] < pents5Index)
    else if (eSP[i] >= pents5Index and eSP[i] < pents6Index)
    {
        if (pents[eSP[i]-pents5Index][0] == p1)
            positionP1 = 0;
        else if (pents[eSP[i]-pents5Index][1] == p1)
            positionP1 = 1;
        else if (pents[eSP[i]-pents5Index][2] == p1)
            positionP1 = 2;
        else if (pents[eSP[i]-pents5Index][3] == p1)
            positionP1 = 3;
        else if (pents[eSP[i]-pents5Index][4] == p1)
            positionP1 = 4;

        if (pents[eSP[i]-pents5Index][0] == p2)
            positionP2 = 0;
        else if (pents[eSP[i]-pents5Index][1] == p2)
            positionP2 = 1;
        else if (pents[eSP[i]-pents5Index][2] == p2)
            positionP2 = 2;
        else if (pents[eSP[i]-pents5Index][3] == p2)
            positionP2 = 3;
        else if (pents[eSP[i]-pents5Index][4] == p2)
            positionP2 = 4;

        if (pents[eSP[i]-pents5Index][0] == p3)
            positionP3 = 0;
        else if (pents[eSP[i]-pents5Index][1] == p3)
            positionP3 = 1;
```

```
                else if (pents[eSP[i]-pents5Index][2] == p3)
                    positionP3 = 2;
                else if (pents[eSP[i]-pents5Index][3] == p3)
                    positionP3 = 3;
                else if (pents[eSP[i]-pents5Index][4] == p3)
                    positionP3 = 4;

                if (positionP1 >= 0 and positionP2 >= 0 and positionP3 >= 0)
                {
                    *neighborElement = eSP[i];
                    if (positionP1 != 4 and positionP2 != 4 and positionP3 != 4 &&
                        positionP1 != 3 and positionP2 != 3 and positionP3 != 3)
                    {
                        *neighborElementFace = 0;
                        done = 1;
                    }
                    else if (positionP1 != 0 and positionP2 != 0 and positionP3 != 0 &&
                        positionP1 != 3 and positionP2 != 3 and positionP3 != 3)
                    {
                        *neighborElementFace = 1;
                        done = 1;
                    }
                    else if (positionP1 != 0 and positionP2 != 0 and positionP3 != 0 &&
                        positionP1 != 1 and positionP2 != 1 and positionP3 != 1)
                    {
                        *neighborElementFace = 3;
                        done = 1;
                    }
                    else if (positionP1 != 1 and positionP2 != 1 and positionP3 != 1 &&
                        positionP1 != 4 and positionP2 != 4 and positionP3 != 4)
                    {
                        *neighborElementFace = 4;
                        done = 1;
                    }
                } // End if (positionP1 >= 0 and positionP2 >= 0 and positionP3 >= 0)
            } // End else if (eSP[i] >= pents5Index and eSP[i] < pents6Index)
        } // End if (eSP[i] != insideThisElement)
        i++; // Increment to the next element id in the list.
    } // End while loop to cycle for a match.
} // End commonElement routine.
```

APPENDIX E

ALGORITHM TO CREATE VOLUME CHUNKING

```
void createVolumeChunking( int numXDivisions, // Number of X subdivisions.
                           int numYDivisions, // Number of Y subdivisions.
                           int numZDivisions ) // Number of Z subdivisions.
{
    int         i                = 0;      // i is used as a counting variable.
    int         j                = 0;      // j is used as a counting variable.
    int         k                = 0;      // k is used as a counting variable.
    int         numVolumes       = numXDivisions*numYDivisions*numZDivisions;
    int         vIndex           = 0;      // Used as an index into chunking structure.
    int         p1               = 0;      // Used to contain the index for point 1.
    int         p2               = 0;      // Used to contain the index for point 2.
    int         p3               = 0;      // Used to contain the index for point 3.
    int         p4               = 0;      // Used to contain the index for point 4.
    int         p1XIndex         = 0;      // The X direction index for point 1.
    int         p1YIndex         = 0;      // The Y direction index for point 1.
    int         p1ZIndex         = 0;      // The Z direction index for point 1.
    INT_1D      *tetsInVolume    = NULL;   // Contains element indices into chunks.
    INT_1D      *tNumTetsInVolume = NULL;  // Temp for constructing numTetsInVolume.
    INT_1D      *numTetsInVolume = NULL;   // Contains number of tets in chunk.
    double      xInc             = 0.0;    // xInc is the increment in the X direction.
    double      yInc             = 0.0;    // yInc is the increment in the Y direction.
    double      zInc             = 0.0;    // zInc is the increment in the Z direction.
    DOUBLE_3D Min;                         // Contains min bounding box information.
    DOUBLE_3D Max;                         // Contains max bounding box information.
    getBoundingBox(Min,Max);                   // Get the bounding box extremes.

    xInc = (Max[0]-Min[0])/(numXDivisions*1.0);     // Calculate the x increment.
    yInc = (Max[1]-Min[1])/(numYDivisions*1.0);     // Calculate the y increment.
    zInc = (Max[2]-Min[2])/(numZDivisions*1.0);     // Calculate the z increment.

    numTetsInVolume = new INT_1D[numVolumes+1]; // Allocate memory.
    tNumTetsInVolume = new INT_1D[numVolumes];  // Allocate temporary array.
    for (i = 0; i < numVolumes; i++)            // Cycle over all volumes.
        tNumTetsInVolume[i] = 0;                // Initialize all in array to 0.

    // Cycle over all tetrahedra in the volume grid.
    for (i = 0; i < numberOfVolumeTetrahedra; i++)
    {
        p1 = tetrahedra[i][0];      // Point index for P1.
        p2 = tetrahedra[i][1];      // Point index for P2.
        p3 = tetrahedra[i][2];      // Point index for P3.
        p4 = tetrahedra[i][3];      // Point index for P4.
```

```
    // Calculate the X index for point P1.
    p1XIndex = (int)(((coordinates[p1][0]-Min[0])/(Max[0]-Min[0]))*numXDivisions);
    // Calculate the Y index for point P1.
    p1YIndex = (int)(((coordinates[p1][1]-Min[1])/(Max[1]-Min[1]))*numYDivisions);
    // Calculate the Z index for point P1.
    p1ZIndex = (int)(((coordinates[p1][2]-Min[2])/(Max[2]-Min[2]))*numZDivisions);
    // If the calculated X index is at the edge of the volume, decrement it.
    if (p1XIndex == numXDivisions)
        p1XIndex--;

    // If the calculated Y index is at the edge of the volume, decrement it.
    if (p1YIndex == numYDivisions)
        p1YIndex--;
    // If the calculated Z index is at the edge of the volume, decrement it.
    if (p1ZIndex == numZDivisions)
        p1ZIndex--;

    // Calculate the index into the volume chunking structure.
    vIndex = p1XIndex*numYDivisions*numZDivisions +
             p1YIndex*numZDivisions +
             p1ZIndex;

    tNumTetsInVolume[vIndex]++; // Increment num tets in volume.
} // End cycle over all tetrahedra in input grid.

Initialize the first value in the array to 0.
numTetsInVolume[0] = 0;
// Set the values in the actual array in the manner similar to
// nESP shown in the construction of elements surrounding a point.
for (i = 1; i <= numVolumes; i++)
    numTetsInVolume[i] = numTetsInVolume[i-1] +
                         tNumTetsInVolume[i-1];
// Deallocate the memory just used in construction of numTetsInVolume.
delete[] tNumTetsInVolume;

// Allocate the actual memory that is the volume chunking structure.
tetsInVolume = new INT_1D[numberOfVolumeTetrahedra];
// Cycle over all tetrahedra in the volume grid.
for (i = 0; i < numberOfVolumeTetrahedra; i++)
{
    p1 = tetrahedra[i][0];      // Point index for P1.
    p2 = tetrahedra[i][1];      // Point index for P2.
    p3 = tetrahedra[i][2];      // Point index for P3.
    p4 = tetrahedra[i][3];      // Point index for P4.
```

```
        // Calculate the X index for point P1.
        p1XIndex = (int)(((coordinates[p1][0]-Min[0])/(Max[0]-Min[0]))*numXDivisions);
        // Calculate the Y index for point P1.
        p1YIndex = (int)(((coordinates[p1][1]-Min[1])/(Max[1]-Min[1]))*numYDivisions);
        // Calculate the Z index for point P1.
        p1ZIndex = (int)(((coordinates[p1][2]-Min[2])/(Max[2]-Min[2]))*numZDivisions);
        // If the calculated X index is at the edge of the volume, decrement it.
        if (p1XIndex == numXDivisions)
            p1XIndex--;
        // If the calculated Y index is at the edge of the volume, decrement it.
        if (p1YIndex == numYDivisions)
            p1YIndex--;
        // If the calculated Z index is at the edge of the volume, decrement it.
        if (p1ZIndex == numZDivisions)
            p1ZIndex--;

        // Calculate the index into the volume chunking structure.
        vIndex = p1XIndex*numYDivisions*numZDivisions +
                 p1YIndex*numZDivisions +
                 p1ZIndex;
        // Place this element i into the chunking structure indicated by vIndex.
        tetsInVolume[numTetsInVolume[vIndex]] = i;
        // Increment the number of elements in this volume chunk by one.
        numTetsInVolume[vIndex]++;
    } // End cyle over all tetrahedra in input grid.
} // End createVolumeChunking.
```

APPENDIX F

ALGORITHM TO GET ELEMENT CONTAINING A POINT

```
void getElementContainingPoint( double x, // x value of the point.
                                double y, // y value of the point.
                                double z, // z value of the point.
                                int *e, // Contains the element found upon return.
                                    // May also contain starting element for search.
                                INT_1D *visited, // Contains visit info for each element.
                                int visit, // Current visit id.
                                Boolean useChunking, // Flag to tell routine that
                                                     // volume chunking used to set
                                                     // starting element.
                                int bruteForce ) // Flag to tell routine that if element
                                                 // not found in search, use brute force
                                                 // searching to cycle through all elements
                                                 // not visited.
{
    int         xIndex = 0;     // The xIndex for the given point.
    int         yIndex = 0;     // The yIndex for the given point.
    int         zIndex = 0;     // The zIndex for the given point.
    int         vIndex = 0;     // vIndex is the index into volume chunking.
    int         sIndex = 0;     // sIndex is the index into numTetsInVolume.
    int         eIndex = 0;     // eIndex is the index into numTetsInVolume.
    int         tetIndex= 0;    // Place holder for possible containing element.
    double      V1     = 0.0;   // Sub volume 1 of candidate tetrahedra.
    double      V2     = 0.0;   // Sub volume 2 of candidate tetrahedra.
    double      V3     = 0.0;   // Sub volume 3 of candidate tetrahedra.
    double      V4     = 0.0;   // Sub volume 4 of candidate tetrahedra.

    DOUBLE_3D  Min;             // Minimum value of bounding volume.
    DOUBLE_3D  Max;             // Maximum value of bounding volume.
    // The volume chunking is based on the bounding box of all input data.
    getBoundingBox(Min,Max);

    if (useChunking)
    {
        // Calculate the x index for the x value sent into the search.
        xIndex = (int)(((x-Min[0])/(Max[0]-Min[0]))*numXDivisions);
        // Calculate the y index for the y value sent into the search.
        yIndex = (int)(((y-Min[1])/(Max[1]-Min[1]))*numYDivisions);
        // Calculate the z index for the z value sent into the search.
        zIndex = (int)(((z-Min[2])/(Max[2]-Min[2]))*numZDivisions);
        // If the xIndex is at the edge of the volume, decrement it.
        if (xIndex == numXDivisions)
            xIndex--;
```

```
    // If the yIndex is at the edge of the volume, decrement it.
    if (yIndex == numYDivisions)
        yIndex--;
    // If the zIndex is at the edge of the volume, decrement it.
    if (zIndex == numZDivisions)
        zIndex--;

    // Calculate the volume index into the volume chunking structure.
    vIndex = xIndex*numYDivisions*numZDivisions +
             yIndex*numZDivisions +
             zIndex;

    // Initialize start and end indices to be zero.
    sIndex = 0;
    eIndex = 0;

    // If the volume chunking index is zero, then sIndex is zero.
    if (vIndex == 0)
        sIndex = 0;
    // Else, find the start index for the elements in this volume chunk.
    else
        sIndex = numTetsInVolume[vIndex-1];

    // The ending index is the value at vIndex.
    eIndex = numTetsInVolume[vIndex];
    // If sIndex and eIndex are the same, then there are no elements in this chunk.
    if (eIndex == sIndex)
        *e = 0;
    // Else, set a candidate element to the first element listed in the chunk.
    else
        *e = tetsInVolume[sIndex];
}

// element is used as a placeholder for the current element index.
element = *e;

// Call the recursive search algorithm with this candidate element index.
// If useChunking is False, then the recursive search starts with an
// element that is passed in through e.
recursiveSearch(x,y,z,element,e,&found,visited,visit,tolerance);

// If the containing element has not been found and bruteForce flag is True,
if (found == 0 and bruteForce)
{
    // Calculate the x index for the x value sent into the search.
```

```
xIndex = (int)(((x-Min[0])/(Max[0]-Min[0]))*numXDivisions);
// Calculate the y index for the y value sent into the search.
yIndex = (int)(((y-Min[1])/(Max[1]-Min[1]))*numYDivisions);
// Calculate the z index for the z value sent into the search.
zIndex = (int)(((z-Min[2])/(Max[2]-Min[2]))*numZDivisions);

// If the xIndex is at the edge of the volume, decrement it.
if (xIndex == numXDivisions)
    xIndex--;
// If the yIndex is at the edge of the volume, decrement it.
if (yIndex == numYDivisions)
    yIndex--;
// If the zIndex is at the edge of the volume, decrement it.
if (zIndex == numZDivisions)
    zIndex--;

// Calculate the volume index into the volume chunking structure.
vIndex = xIndex*numYDivisions*numZDivisions +
         yIndex*numZDivisions +
         zIndex;

// Initialize start and end indices to be zero.
sIndex = 0;
eIndex = 0;

// If the volume chunking index is zero, then sIndex is zero.
if (vIndex == 0)
    sIndex = 0;
// Else, find the start index for the elements in this volume chunk.
else
    sIndex = numTetsInVolume[vIndex-1];
// The ending index is the value at vIndex.

eIndex = numTetsInVolume[vIndex];

// Cycle over all elements in this volume chunk.
for (i = sIndex; i < eIndex; i++)
{
    // Dereference the index for this element in this chunk.
    tetIndex = tetsInVolume[i];


    // Check this element only if it has not already been visited.
    if (visited[tetIndex] != visit)
```

```
            {
                // Set the visited flag for this element.
                visited[tetIndex] = visit;

                // Check whether this element contains the given point.
                found = doesElementContainThisPoint(x,y,z,tetIndex,tolerance,
                                                    &V1,&V2,&V3,&V4);
                if (found)
                {
                    *e = tetIndex;
                    return;
                }
            } // End if (visited[tetIndex] != visit)
        } // End for (i = sIndex; i < eIndex; i++)
        *e = -999;
    } // End if (found == 0 and bruteForce)
    else if (found == 0 and !bruteForce)
    *e = -999;
} // End getElementContainingPoint.
```

```
void recursiveSearch( double x, // x value of point given.
                      double y, // y value of point given.
                      double z, // z value of point given.
                      int element, // element currently inside.
                      int *e, // return value for element found.
                      int *found, // returns whether element found.
                      INT_1D *visited, // array containing visited flags for each element.
                      int visit, // current visit id.
                      double tolerance ) // tolerance for element containment.
{
    double    V1         = 0.0;   // Sub volume one of the given tetrahedra.
    double    V2         = 0.0;   // Sub volume two of the given tetrahedra.
    double    V3         = 0.0;   // Sub volume three of the given tetrahedra.
    double    V4         = 0.0;   // Sub volume four of the given tetrahedra.
    Boolean   foundFlag  = False; // Flag to determine whether element found.
    // Set the visited flag for this element to the current visit id.
    visited[element] = visit;
    // Check whether this element contains the given point.
    foundFlag = doesElementContainThisPoint(x,y,z,element,tolerance,
                                            &V1,&V2,&V3,&V4);
    // If this element contains the given point, we are done.
    if (foundFlag)
    {
        *e = element;
        *found = 1;
        return;
    }
    // Else, we have to go through each of the neighbors connected to this element.
    else
    {
        // If the calculated sub volume is negative and the element has not been found,
        if ( V1 < 0.0 and *found != 1)
        {
            // If neighbor 2 exists and it has not already been visited,
            if ( eN[element][1] >= 0 and visited[eN[element][1]] != visit )
            {
                // Call the recursive searching routine with element neighbor 2.
                recursiveSearch( x,
                                 y,
                                 z,
                                 eN[element][1],
                                 e,
                                 found,
                                 visited,
                                 visit,
```

```
                    tolerance );
    }
}

// If the calculated sub volume is negative and the element has not been found,
if ( V2 < 0.0 and *found != 1)
{
    // If neighbor 3 exists and it has not already been visited,
    if ( eN[element][2] >= 0 and visited[eN[element][2]] != visit )
    {
        // Call the recursive searching routine with element neighbor 3.
        recursiveSearch( x,
                         y,
                         z,
                         eN[element][2],
                         e,
                         found,
                         visited,
                         visit,
                         tolerance );
    }
}

// If the calculated sub volume is negative and the element has not been found,
if ( V3 < 0.0 and *found != 1)
{
    // If neighbor 4 exists and it has not already been visited,
    if ( eN[element][3] >= 0 and visited[eN[element][3]] != visit )
    {
        // Call the recursive searching routine with element neighbor 4.
        recursiveSearch( x,
                         y,
                         z,
                         eN[element][3],
                         e,
                         found,
                         visited,
                         visit,
                         tolerance );
    }
}
```

```
        // If the calculated sub volume is negative and the element has not been found,
        if ( V4 < 0.0 and *found != 1)
        {
            // If neighbor 0 exists and it has not already been visited,
            if ( eN[element][0] >= 0 and visited[eN[element][0]] != visit )
            {
                // Call the recursive searching routine with element neighbor 0.
                recursiveSearch( x,
                                 y,
                                 z,
                                 eN[element][0],
                                 e,
                                 found,
                                 visited,
                                 visit,
                                 tolerance );
            }
        }
    } // End else.
} // End recursiveSearch.
```

```
Boolean doesElementContainThisPoint( double x,
                                     double y,
                                     double z,
                                     int element,
                                     double tolerance,
                                     double *V1,
                                     double *V2,
                                     double *V3,
                                     double *V4 )
{
    int     p1      = 0;    // p1 is the point 1 index.
    int     p2      = 0;    // p2 is the point 2 index.
    int     p3      = 0;    // p3 is the point 3 index.
    int     p4      = 0;    // p4 is the point 4 index.
    double  V       = 0.0;  // V is the volume of the element.
    double  volmin  = 0.0;  // volmin holds a minimum volume.
    double  w       = 0.0;  // w is a weighting factor.
    p1   = tetrahedra[element][0];
    p2   = tetrahedra[element][1];
    p3   = tetrahedra[element][2];
    p4   = tetrahedra[element][3];

    // Calculate the volume of the tetrahedra indicated by element.
    V    = calculateVolume(coordinates[p1],
                           coordinates[p2],
                           coordinates[p3],
                           coordinates[p4]);
    // V1 is the volume created by the given point, point P2, point P3, and point P4.
    *V1  = calculateVolume(X,coordinates[p2],coordinates[p3],coordinates[p4]);

    // V2 is the volume created by the given point, point P1, point P4, and point P3.
    *V2  = calculateVolume(X,coordinates[p1],coordinates[p4],coordinates[p3]);

    // V3 is the volume created by the given point, point P4, point P1, and point P2.
    *V3  = calculateVolume(X,coordinates[p4],coordinates[p1],coordinates[p2]);

    // Because V1+V2+V3+V4 must be 1.0, then V4 is given as,
    *V4  = 1.0 - (*V1) - (*V2) - (*V3);

    // Find the minimum of all of the subvolumes V1, V2, V3, and V4.
    volmin = MIN ((*V1), MIN ((*V2), MIN ((*V3), (*V4))));

    // Weight this by a tolerance multiplied by the volume of the element.
    w      = volmin + tolerance * V;
```

```
    if (w >= 0.0)       // If the weighting is positive,
        return True;    // this element contains the given point.
    else                // else,
        return False;   // this element does not contain the given point.
} // End doesElementContainThisPoint.
```

APPENDIX G

ALGORITHM TO CALCULATE AN ARBITRARY CUTTING PLANE WITH NO DUPLICATE POINTS

```
void calculateCuttingPlane( DOUBLE_3D point, // A point on the cutting plane.
                            DOUBLE_3D normal, // The normal to the cutting plane.
                            double tolerance, // A tolerance for intersections.
                            int *nAllocatedSurface, // Current size of Tris.
                            int *nPts, // Number of points in extraced surface.
                            DOUBLE_3D **Pts, // List of resulting points.
                            int *nTris, // Number of triangles in extraced surface.
                            INT_3D **Tris, // List of resulting triangle point indices.
                            INT_1D **EFPts, // List of element ids that correspond
                                            // to points in extracted surface.
                            int gridIndex, // If dealing with multiple grids,
                                           // then gridIndex will vary.
                            INT_1D *nAllocatedElement, // Current size of the array Elem.
                            INT_1D **nElem, // Number of intersected elements in grid
                                            // indicated by gridIndex.
                            INT_1D **Elem ) // List of intersected element ids in
                                            // extracted surface.
{
    Boolean  f1  = False;  // Indicates whether face 1 of the current
                           // element has been intersected.
    Boolean  f2  = False;  // Indicates whether face 2 of the current
                           // element has been intersected.
    Boolean  f3  = False;  // Indicates whether face 3 of the current
                           // element has been intersected.
    Boolean  f4  = False;  // Indicates whether face 4 of the current
                           // element has been intersected.

    Boolean *eVisited  = NULL; // Array to contain whether element has been
                               // visited in surface extraction routine.
    int      nElements = 0;    // The number of elements
    int      i         = 0;    // i is used as a counting variable.
    int      j         = 0;    // j is used as a counting variable.
    int      k         = 0;    // k is used as a counting variable.
    int      e         = 0;    // Contains local element id.
    int      pIndex    = 0;    // pIndex contains a point index.
    int      eIndex    = 0;    // eIndex contains an element index.
    int      inPts     = 0;    // The local number of intersection points per element.
    int      inTris    = 0;    // The local number of triangles generated from element
                               // plane intersection.
    int      it[6];            // Contains whether each edge of the tetrahedra has
                               // been intersected or not.
    int      pit[6];           // Contains the intersection point on each edge of the
                               // tetrahedra.
    int      gIndex[6];        // Contains global index of intersection point in the
                               // array Pts.
```

```
int       pgIndex[6];          // Contains previous intersection points global index in the
                               // array Pts.
INT_1D  *visited    = NULL;    // Array to contain whether element has been
                               // visited for use in point containment.
INT_1D  *eit        = NULL;    // Contains all elements intersection indicators.
INT_1D  *egIndex    = NULL;    // Contains all elements global index values in the array Pts.
INT_3D  *iTris      = NULL;    // Contains the local triangles cut on a per element basis.


double          t0    = 0.0;      // Variable to determine edge crossings for
                                  // those containing point 0.
double          t1    = 0.0;      // Variable to determine edge crossings for
                                  // those containing point 1.
double          t2    = 0.0;      // Variable to determine edge crossings for
                                  // those containing point 2.
double          t3    = 0.0;      // Variable to determine edge crossings for
                                  // those containing point 3.
double          tol   = 0.0;      // Local cutting tolerance based on tolerance passed
                                  // into routine.
DOUBLE_3D  p0;                    // Variable to contain values for point 0.
DOUBLE_3D  p1;                    // Variable to contain values for point 1.
DOUBLE_3D  p2;                    // Variable to contain values for point 2.
DOUBLE_3D  p3;                    // Variable to contain values for point 3.
DOUBLE_3D  ppt[6];                // Previous elements intersection point.
DOUBLE_3D  pt[6];                 // Local elements intersection points.
DOUBLE_3D  *iPts = NULL;          // Local elements intersection points.
DOUBLE_3D  *ept  = NULL;          // Contains all intersection points for all elements.

// Allocate the memory for recording element visits.
eVisited   = new Boolean[numberOfVolumeTetrahedra];
// Allocate the memory for element containment visits.
visited    = new INT_1D[numberOfVolumeTetrahedra];
// Allocate the memory to record all intersection indicators for all elements.
eit        = new INT_1D[numberOfVolumeTetrahedra*6];
// Allocate the memory to record global index for intersection points for all elements.
egIndex    = new INT_1D[numberOfVolumeTetrahedra*6];
// Allocate the memory to handle local intersections on per element basis.
iTris      = new INT_3D[20];
// Allocated to store local intersection points on per element basis.
iPts       = new DOUBLE_3D[20];
// Allocated to record all intersection points for all elements.
ept        = new DOUBLE_3D[numberOfVolumeTetrahedra*6];

i = 0; // i is used as a global index.
// For all tetrahedra in the input grid, initialize global structures.
```

```
for (j = 0; j < numberOfVolumeTetrahedra*6; j++)
{
    eit[i] = 0;                 // Initialize intersection indicators to 0.
    ept[i][0] = 0.0;            // Initialize x intersection point value to 0.0.
    ept[i][1] = 0.0;            // Initialize y intersection point value to 0.0.
    ept[i][2] = 0.0;            // Initialize z intersection point value to 0.0.
    egIndex[i] = -999;          // Initialize global index to 0.
    i++;                        // Increment the global index.
}

*nPts = 0; // Initialize the number of points to 0.
*nTris = 0; // Initialize the number of triangles to 0.

for (i = 0; i < gridIndex; i++)             // For each grid that is being cut,
    nElements += (*nElem)[gridIndex];       // find the total number of elements.

// Cycle over all tetrahedra in the input grid and initialize visits.
for (i = 0; i < numberOfVolumeTetrahedra; i++)
{
    visited[i] = 0;             // Initialize visited for searching all to visit 0.
    eVisited[i] = False;        // Initialize all elements to not visited.
}

getElementContainingPoint( point[0], // X value of point on the cutting plane.
                           point[1], // Y value of point on the cutting plane.
                           point[2], // Z value of point on the cutting plane.
                           &e, // Element containing this point is returned in e.
                           visited, // Contains current visits for each element.
                           1, // The visit flag for this round of searching.
                           1E-5, // Provide a tolerance for element containment.
                           True, // Get a starting element using chunking scheme.
                           True ); // If point is not found locally, search globally.

p0[0] = coordinates[tetrahedra[e][0]][0];  // Set x value of point 0.
p0[1] = coordinates[tetrahedra[e][0]][1];  // Set y value of point 0.
p0[2] = coordinates[tetrahedra[e][0]][2];  // Set z value of point 0.

p1[0] = coordinates[tetrahedra[e][1]][0];  // Set x value of point 1.
p1[1] = coordinates[tetrahedra[e][1]][1];  // Set y value of point 1.
p1[2] = coordinates[tetrahedra[e][1]][2];  // Set z value of point 1.


p2[0] = coordinates[tetrahedra[e][2]][0];  // Set x value of point 2.
p2[1] = coordinates[tetrahedra[e][2]][1];  // Set y value of point 2.
p2[2] = coordinates[tetrahedra[e][2]][2];  // Set z value of point 2.
```

```
p3[0] = coordinates[tetrahedra[e][3]][0];  // Set x value of point 3.
p3[1] = coordinates[tetrahedra[e][3]][1];  // Set y value of point 3.
p3[2] = coordinates[tetrahedra[e][3]][2];  // Set z value of point 3.

t0 = (p0[0] - point[0]) * normal[0] +     // Compute the dot product
     (p0[1] - point[1]) * normal[1] +     // of vector from planar point to
     (p0[2] - point[2]) * normal[2];      // point 0 and the normal to the plane.

t1 = (p1[0] - point[0]) * normal[0] +     // Compute the dot product
     (p1[1] - point[1]) * normal[1] +     // of vector from plane point to
     (p1[2] - point[2]) * normal[2];      // point 0 and the normal to the plane.

t2 = (p2[0] - point[0]) * normal[0] +     // Compute the dot product
     (p2[1] - point[1]) * normal[1] +     // of vector from plane point to
     (p2[2] - point[2]) * normal[2];      // point 0 and the normal to the plane.

t3 = (p3[0] - point[0]) * normal[0] +     // Compute the dot product
     (p3[1] - point[1]) * normal[1] +     // of vector from plane point to
     (p3[2] - point[2]) * normal[2];      // point 0 and the normal to the plane.

// Base tolerance on the max edge length and machine precision.
tol = (getMaxEdgeLength(e))*tolerance;

// If the cutting plane intersects this tetrahedra,
if ( (t0 >= -tol and t1 <= tol) or (t0 <= tol and t1 >= -tol) or
     (t0 >= -tol and t2 <= tol) or (t0 <= tol and t2 >= -tol) or
     (t0 >= -tol and t3 <= tol) or (t0 <= tol and t3 >= -tol) or
     (t1 >= -tol and t2 <= tol) or (t1 <= tol and t2 >= -tol) or
     (t1 >= -tol and t3 <= tol) or (t1 <= tol and t3 >= -tol) or
     (t2 >= -tol and t3 <= tol) or (t2 <= tol and t3 >= -tol) )
{
     for (i = 0; i < 6; i++)          // For all potential intersection points,
     {
          pt[i][0] = 0.0;             // Initialize x intersection to 0.0.
          pt[i][1] = 0.0;             // Initialize y intersection to 0.0.
          pt[i][2] = 0.0;             // Initialize z intersection to 0.0.
          it[i] = 0;                  // Initialize intersection indicator to 0.
          gIndex[i] = 0;              // Initialize global index values to 0.

          ppt[i][0] = 0.0;            // Initialize previous x intersection to 0.0.
          ppt[i][1] = 0.0;            // Initialize previous x intersection to 0.0.
          ppt[i][2] = 0.0;            // Initialize previous x intersection to 0.0.
          pit[i] = 0;                 // Initialize previous intersection indicator to 0.
          pgIndex[i] = 0;             // Initialize previous global index values to 0.
```

```
}
// Calculate the intersection with the plane and the tetrahedra.
intersectPlaneWithTetrahedra( e,          // The element id for the tetrahedra.
                              point,      // The point on the cutting plane.
                              normal,     // The normal to the cutting plane.
                              p0,         // Point 0 on the tetrahedra.
                              p1,         // Point 1 on the tetrahedra.
                              p2,         // Point 2 on the tetrahedra.
                              p3,         // Point 3 on the tetrahedra.
                              t0,         // Cut parameter for point 0.
                              t1,         // Cut parameter for point 1.
                              t2,         // Cut parameter for point 2.
                              t3,         // Cut parameter for point 3.
                              &f1,        // Returns whether face 1 is cut or not.
                              &f2,        // Returns whether face 2 is cut or not.
                              &f3,        // Returns whether face 3 is cut or not.
                              &f4,        // Returns whether face 4 is cut or not.
                              tolerance,  // Used to determine whether cut or not.
                              &inPts,     // Local number of intersection points.
                              &iPts,      // Local intersection points.
                              &inTris,    // Local number of intersection triangles.
                              &iTris,     // Local intersection triangles.
                              0,          // Current index into the array containing
                                          // all intersection points.
                              pit,        // Previous elements intersection indicators.
                              ppt,        // Previous elements int. point values.
                              pgIndex,    // Global index of previous intersections.
                              it,         // Current elements intersection indicators
                                          // (to be determined).
                              pt,         // Current elements int. point values
                                          // (to be determined).
                              gIndex,     // Current global index in global pts. array.
                              False,      // Flag to remove duplicate int.
                              eN,         // Element neighbor array.
                              -1,         // Previous element index.
                              -1,         // Face id from prev. element.
                              eit,        // Int. indicators for all cut elements.
                              ept,        // Int. point values for all cut elements.
                              egIndex ); // Global pt. indices for all cut elements.

// If recording the current element information would exceed allocated memory,
// then reallocate more memory.
if (*nAllocatedElement == nElements)
    reallocate memory
```

```
(*Elem)[nElements++] = e;       // Store the element that has just been intersected.
(*nElem)[gridIndex]++;          // Increment the number of elements for the grid that
                                // it belongs to.

// If recording the current triangle information would exeed allocated memory,
// then reallocate more memory.
if (*nAllocatedSurface == inTris)
    reallocate memory

pIndex = *nPts;                 // Store the current index as of now.
for (i = 0; i < inPts; i++)     // For all intersection points just found,
{
    (*Pts)[*nPts][0] = iPts[i][0]; // Store the x value of the intersection point.
    (*Pts)[*nPts][1] = iPts[i][1]; // Store the y value of the intersection point.
    (*Pts)[*nPts][2] = iPts[i][2]; // Store the z value of the intersection point.
    (*EFPts)[*nPts] = e;        // Store the element that has just been cut.
    (*nPts)++;                  // Increment number of points in the cutting plane.
}

for (i = 0; i < inTris; i++)        // For all triangles formed from the cut,
{
    (*Tris)[*nTris][0] = iTris[i][0]; // Store first point index.
    (*Tris)[*nTris][1] = iTris[i][1]; // Store second point index.
    (*Tris)[*nTris][2] = iTris[i][2]; // Store third point index.
    (*nTris)++;                     // Increment the number of triangles in the
                                    // cutting plane.
}

// Find the cut elements index in the global element array.
eIndex = e*6;

// For all possible intersection points,
for (i = 0; i < 6; i++)
{
    // If this intersection indicator says edge has been cut,
    if (it[i])
    {
        ept[eIndex][0] = pt[i][0]; // Store the x point value.
        ept[eIndex][1] = pt[i][1]; // Store the y point value.
        ept[eIndex][2] = pt[i][2]; // Store the z point value.
        eit[eIndex] = it[i];       // Store the intersection indicator.

        // Store the global index for the point into the Pts array.
        egIndex[eIndex] = gIndex[i];
```

```
            ppt[i][0] = pt[i][0]; // Make this x value the previous x value.
            ppt[i][1] = pt[i][1]; // Make this y value the previous y value.
            ppt[i][2] = pt[i][2]; // Make this z value the previous z value.
            pit[i] = it[i];       // Make this intersection indicator the previous
                                  // intersection indicator.

            // Make this global index the previous global index.
            pgIndex[i] = gIndex[i] = pIndex;
            pIndex++;     // Increment the current index into Pts.
        }
        eIndex++;     // Increment index into global information array.
    }
    eVisited[e] = True;  // Set this element as having been visited.
    recursivelyCalculateCut( point, // The point in the cutting plane.
                             normal, // The normal to the cutting plane.
                             tolerance, // User defined tolerance passed in originally.
                             nAllocatedSurface, // The size of the Tris array right now.
                             nPts, // Contains the number of points currently in cut.
                             Pts, // Contains the points currently in the cut.
                             nTris, // Contains the number of triangles currently in cut.
                             Tris, // Contains the triangles currently in the cut.
                             EFPts, // Contains element index for each point in the cut.
                             nAllocatedElement, // The size of the Elem array right now.
                             nElem, // Number of elements that have been currently cut.
                             Elem, // The elements that have been currently cut.
                             e, // The element that the algorithm has just cut.
                             f1, // Flag indicating whether face 1 of e has been cut.
                             f2, // Flag indicating whether face 2 of e has been cut.
                             f3, // Flag indicating whether face 3 of e has been cut.
                             f4, // Flag indicating whether face 4 of e has been cut.
                             eVisited, // Contains whether elements have been visited.
                             inPts, // Contains number of local intersection points.
                             iPts, // Contains local intersection points.
                             inTris, // Contains number of local intersection triangles.
                             iTris, // Contains local intersection triangles.
                             gridIndex, // Contains the number of grids being cut.
                             &nElements, // The total number of elements in all grids.
                             ppt, // Previous intersection points.
                             pit, // Previous intersection indicators.
                             pgIndex, // Prev. global indices in Pts for int. pts.
                             pt, // Current intersection points.
                             it, // Current intersection indicators.
                             gIndex, // Current global indices in Pts for int. pts.
                             ept, // Contains all int. pts. for cut elements.
                             eit, // Contains all int. indicators for cut elements.
```

```
                                    egIndex ); // Contains all global indices into Pts for int. pts.
    } // End if tetrahedra intersects the cutting plane.
} // End calculateCuttingPlane.
```

```
void recursivelyCalculateCut( DOUBLE_3D point, // The point on the plane.
                              DOUBLE_3D normal, // The normal to the cutting plane.
                              double tolerance, // User defined tolerance passed in originally.
                              int *nAllocatedSurface, // The size of the Tris array right now.
                              int *nPts, // Contains number of points currently in the cut.
                              DOUBLE_3D **Pts, // Contains the points currently recorded.
                              int *nTris, // The number of triangles currently in the cut.
                              INT_3D **Tris, // Contains the triangles currently in the cut.
                              INT_1D **EFPts, // Contains element indices for each cut pt.
                              INT_1D *nAllocatedElement, // The size of the Elem array now.
                              INT_1D **nElem, // Number of elements in the current cut.
                              INT_1D **Elem, // Elements in the current cut.
                              int e, // Current element.
                              Boolean f1, // Flag indicating whether face 1 of e has been cut.
                              Boolean f2, // Flag indicating whether face 2 of e has been cut.
                              Boolean f3, // Flag indicating whether face 3 of e has been cut.
                              Boolean f4, // Flag indicating whether face 4 of e has been cut.
                              Boolean *visited, // Allocated in auxiliary for element visited.
                              int inPts, // Contains number of local intersection points.
                              DOUBLE_3D *iPts, // Contains local intersection points.
                              int inTris, // Contains number of local intersection triangles.
                              INT_3D *iTris, // Contains local intersection triangles.
                              int gridIndex, // Number of grids being cut.
                              int *nElements, // The total number of elements in all grids.
                              DOUBLE_3D ppt[6], // Previous elements intersection points.
                              INT_1D pit[6], // Previous elements intersection indicators.
                              INT_1D pgIndex[6], // Previous elements global point indices.
                              DOUBLE_3D pt[6], // Current intersection points.
                              INT_1D it[6], // Current intersection indicators.
                              INT_1D gIndex[6], // Current global point indices.
                              DOUBLE_3D ept[6], // Global intersection point information.
                              INT_1D eit[6], // Global intersection indicator information.
                              INT_1D egIndex[6] ) // Global index information.
{
    Boolean     ef1       = False;  // Local variable for whether face 1 cut.
    Boolean     ef2       = False;  // Local variable for whether face 2 cut.
    Boolean     ef3       = False;  // Local variable for whether face 3 cut.
    Boolean     ef4       = False;  // Local variable for whether face 4 cut.
    int         i         = 0;      // Local counting variable.
    int         pIndex    = 0;      // Local point index variable.
    int         eIndex    = 0;      // Local element index variable.
    INT_1D      spit[6];            // Local intersection points.
    INT_1D      spgIndex[6];        // Local previous global index information.
    INT_1D      sit[6];             // Local intersection indicators.
    INT_1D      sgIndex[6];         // Local global index information.
```

```
double      t0      = 0.0;   // Local intersection parameter for point 0.
double      t1      = 0.0;   // Local intersection parameter for point 1.
double      t2      = 0.0;   // Local intersection parameter for point 2.
double      t3      = 0.0;   // Local intersection parameter for point 3.
double      tol     = 0.0;   // Local weighted tolerance.
DOUBLE_3D   p0;              // Local point 0 variable.
DOUBLE_3D   p1;              // Local point 1 variable.
DOUBLE_3D   p2;              // Local point 2 variable.
DOUBLE_3D   p3;              // Local point 3 variable.
DOUBLE_3D   sppt[6];         // Local previous point intersection information.
DOUBLE_3D   spt[6];          // Local current point intersection information.

if (f1 and eN[e][0] >= 0 and !(visited[eN[e][0]]))
{
    for (i = 0; i < 6; i++)          // For all potential intersections,
    {
        sppt[i][0] = ppt[i][0];      // Record local copy of previous point.
        sppt[i][1] = ppt[i][1];      // Record local copy of previous point.
        sppt[i][2] = ppt[i][2];      // Record local copy of previous point.
        spit[i] = pit[i];            // Record local copy of previous indicator.
        spgIndex[i] = pgIndex[i];    // Record local copy of previous global indices.
        spt[i][0] = pt[i][0];        // Record local copy of current point.
        spt[i][1] = pt[i][1];        // Record local copy of current point.
        spt[i][2] = pt[i][2];        // Record local copy of current point.
        sit[i] = it[i];              // Record local copy of indicator.
        sgIndex[i] = gIndex[i];      // Record local copy of global indices.
    }

    visited[eN[e][0]] = True;        // Indicate this neighboring element has been visited.

    // Set the values for point 1 of face 1 of element e.
    p0[0] = coordinates[tetrahedra[eN[e][0]][0]][0];
    p0[1] = coordinates[tetrahedra[eN[e][0]][0]][1];
    p0[2] = coordinates[tetrahedra[eN[e][0]][0]][2];



    // Set the values for point 2 of face 1 of element e.
    p1[0] = coordinates[tetrahedra[eN[e][0]][1]][0];
    p1[1] = coordinates[tetrahedra[eN[e][0]][1]][1];
    p1[2] = coordinates[tetrahedra[eN[e][0]][1]][2];

    // Set the values for point 3 of face 1 of element e.
    p2[0] = coordinates[tetrahedra[eN[e][0]][2]][0];
    p2[1] = coordinates[tetrahedra[eN[e][0]][2]][1];
```

```
p2[2] = coordinates[tetrahedra[eN[e][0]][2]][2];

// Set the values for point 4 of face 1 of element e.
p3[0] = coordinates[tetrahedra[eN[e][0]][3]][0];
p3[1] = coordinates[tetrahedra[eN[e][0]][3]][1];
p3[2] = coordinates[tetrahedra[eN[e][0]][3]][2];

// Compute the dot product of vector from planar point to point 0
// and the normal to the plane.
t0 = (p0[0] - point[0]) * normal[0] +
     (p0[1] - point[1]) * normal[1] +
     (p0[2] - point[2]) * normal[2];

// Compute the dot product of vector from planar point to point 1
// and the normal to the plane.
t1 = (p1[0] - point[0]) * normal[0] +
     (p1[1] - point[1]) * normal[1] +
     (p1[2] - point[2]) * normal[2];

// Compute the dot product of vector from planar point to point 2
// and the normal to the plane.
t2 = (p2[0] - point[0]) * normal[0] +
     (p2[1] - point[1]) * normal[1] +
     (p2[2] - point[2]) * normal[2];

// Compute the dot product of vector from planar point to point 3
// and the normal to the plane.
t3 = (p3[0] - point[0]) * normal[0] +
     (p3[1] - point[1]) * normal[1] +
     (p3[2] - point[2]) * normal[2];

// Base tolerance on the max edge length and machine precision.
tol = (getMaxEdgeLength(eN[e][0]))*tolerance;

// If the cutting plane intersects this tetrahedra,
if ( (t0 >= -tol and t1 <= tol) or (t0 <= tol and t1 >= -tol) or
     (t0 >= -tol and t2 <= tol) or (t0 <= tol and t2 >= -tol) or
     (t0 >= -tol and t3 <= tol) or (t0 <= tol and t3 >= -tol) or
     (t1 >= -tol and t2 <= tol) or (t1 <= tol and t2 >= -tol) or
     (t1 >= -tol and t3 <= tol) or (t1 <= tol and t3 >= -tol) or
     (t2 >= -tol and t3 <= tol) or (t2 <= tol and t3 >= -tol) )
{
    intersectPlaneWithTetrahedra( eN[e][0],
                                  point,
                                  normal,
```

```
                                p0,
                                p1,
                                p2,
                                p3,
                                t0,
                                t1,
                                t2,
                                t3,
                                &ef1,
                                &ef2,
                                &ef3,
                                &ef4,
                                tolerance,
                                &inPts,
                                &iPts,
                                &inTris,
                                &iTris,
                                *nPts,
                                spit,
                                sppt,
                                spgIndex,
                                sit,
                                spt,
                                sgIndex,
                                True,
                                eN,
                                e,
                                0,
                                eit,
                                ept,
                                egIndex );

// If the memory has been exceeded, then reallocate the memory.
if (*nAllocatedElement == *nElements)
    reallocate memory.

// Record the element information and increment the number of elements.
(*Elem)[(*nElements)++] = eN[e][0];
(*nElem)[gridIndex]++;

// If the surface memory has been exceeded, then reallocate the memory.
if (*nAllocatedSurface == inTris)
    reallocate memory.

int index = 0;              // Index variable.
```

```
int begIndex = *nPts;    // Save starting index for this set of intersection points.

// Cycle over all possible intersections.
for (i = 0; i < 6; i++)
{
    // If an intersection has occurred and the point is not a duplicate,
    if ( sit[i] == 1 and sgIndex[i] >= begIndex )
    {
        (*Pts)[*nPts][0] = iPts[index][0]; // Record the x value in the Pts array.
        (*Pts)[*nPts][1] = iPts[index][1]; // Record the y value in the Pts array.
        (*Pts)[*nPts][2] = iPts[index][2]; // Record the z value in the Pts array.
        (*EFPts)[*nPts] = eN[e][0];        // Record the element for this point.
        (*nPts)++; // Increment the number of actual points in the cutting plane.
        index++;   // Increment the index.
    }
}

// Cycle over all triangles resulting from the intersection.
for (i = 0; i < inTris; i++)
{
    (*Tris)[*nTris][0] = iTris[i][0];  // Record index one of the triangle.
    (*Tris)[*nTris][1] = iTris[i][1];  // Record index two of the triangle.
    (*Tris)[*nTris][2] = iTris[i][2];  // Record index three of the triangle.
    (*nTris)++;                        // Record the number of triangles.
}

eIndex = (eN[e][0])*6;        // Dereference element index.

// Cycle over all possible intersections.
for (i = 0; i < 6; i++)
{
    // If the edge is intersected,
    if (sit[i])
    {
        ept[eIndex][0] = spt[i][0];        // Record the x value.
        ept[eIndex][1] = spt[i][1];        // Record the y value.
        ept[eIndex][2] = spt[i][2];        // Record the z value.
        eit[eIndex] = sit[i];              // Record the intersection indicator.
        egIndex[eIndex] = sgIndex[i];      // Record the global index.
    }
    eIndex++;                              // Increment the index.
}

recursivelyCalculateCut( point, // The point on the cutting plane.
                         normal, // The normal to the cutting plane.
```

```
                                    tolerance, // User defined tolerance.
                                    nAllocatedSurface, // Amt of surface info memory.
                                    nPts, // Number of pts. in the current cutting plane.
                                    Pts, // The points currently in the cutting plane.
                                    nTris, // Number of triangles in the current cutting plane.
                                    Tris, // The triangles currently in the cutting plane.
                                    EFPts, // The elements for each point in the cutting plane.
                                    nAllocatedElement, // Amt of element info memory.
                                    nElem, // The number of elements in the cutting plane.
                                    Elem, // Element indices intersected by the cutting plane.
                                    eN[e][0], // The current element.
                                    ef1, // Whether face 1 of current element is intersected.
                                    ef2, // Whether face 2 of current element is intersected.
                                    ef3, // Whether face 3 of current element is intersected.
                                    ef4, // Whether face 4 of current element is intersected.
                                    visited, // Contains visit information for each element.
                                    inPts, // Local number of intersection points.
                                    iPts, // Local intersection points.
                                    inTris, // Local number of intersected triangles.
                                    iTris, // Local intersected triangles.
                                    gridIndex, // Number of grids being cut.
                                    nElements, // Total number of elements in all grids.
                                    spt, // Previous point intersections.
                                    sit, // Previous intersection indicators.
                                    sgIndex, // Previous global indices.
                                    spt, // Current point intersections.
                                    sit, // Current intersection indicators.
                                    sgIndex, // Current global indices.
                                    ept, // Global point intersection information.
                                    eit, // Global intersection indicator information.
                                    egIndex ); // Global point index information.
        } // End if cutting plane intersects this tetrahedra.
    } // End if (f1 and eN[e][0] >= 0 and !(visited[eN[e][0]]))
    if (f2 and eN[e][1] >= 0 and !(visited[eN[e][1]]))
        // Method analogous to that for face 1.
    if (f3 and eN[e][2] >= 0 and !(visited[eN[e][2]]))
        // Method analogous to that for face 1.
    if (f4 and eN[e][3] >= 0 and !(visited[eN[e][3]]))
        // Method analogous to that for face 1.
} // End recursivelyCalculateCut.
```